



Course Intro & Objectives

CS 2110 **Computer Organization**

An introduction to basic computer hardware, machine language, assembly language, and C programming.

Instructor

- Caleb Southern
 - caleb.southern@gatech.edu
 - Include “2110” in the subject line

- Shawn Wahi (Head TA)

Office Hours and
Contact Information on
the Canvas CS2110 Home
Page

Objectives

- ↗ To understand the structure and operation of a modern computer from the ground up.
- ↗ Understand basic hardware concepts: digital circuits, gates, bits, bytes, number representation
- ↗ Understand the Von Neumann model and the structure and operation of a basic datapath

Objectives

- ↗ Structure and function of machine language instructions
- ↗ Structure and function of a symbolic assembly language
- ↗ Basic concepts of computer systems such as the runtime stack, simple I/O devices
- ↗ Introduce the C language with particular emphasis on the underlying assembly and machine language as well as interaction with hardware.

From the point of continuity

(Kishore Ramachandran)

This sounds like a lot of work!

"Can't we make
this fun, too?"

Classes

- **Lecture**

- **Tue/Thu**

- **Section B – 2:00pm – 3:15pm**

- **Section C – 3:30pm – 4:45pm**

- **We'll have a 10 minute break in the middle of lecture**

- **There is an Attendance Quiz due for most lectures.**

- **Lab (*required*)**

- **Mon/Wed 3:30pm, 5:00pm, or 6:30pm**

- **Mix of tutorial, practice, and evaluation**

- **You are required to attend, attendance is taken.**

- **This is not a supplemental help session**

Textbooks

- ↗ Required
 - ↗ *Introduction to Computing Systems, 3rd edition:*
Patt & Patel
 - ↗ *The C Programming Language:* Kernighan & Ritchie
- ↗ Recommended (If you want a Linux book)
 - ↗ Mastering Linux: Paul Wang

Canvas

- We will be using the Canvas LMS*
- <http://canvas.gatech.edu>
- Used for
 - Assignment distribution,
 - Assignment turn-in (along with Gradescope),
 - Grade display

*LMS - Learning Management System

Assignment Values

Item	Number (approx.)	Totals
Homework	10	30%
Quizzes	4	20%
Timed Labs	4	20%
Lecture Attendance		2%
Lab Attendance		3%
Final Exam	1	25%
TOTAL		100%

Homework

- Usually every week
- Types of assignments
 - Logic Simulation
 - Machine Language programming
 - Assembly programming
 - C programming
- High-Level Collaboration is allowed on Homework
 - You can share ideas ***but not source code!***

Homework

- Even though it looks like each homework doesn't count for many points nothing could be further from the truth!
- You cannot and will not do well in lab and on tests if you do not have a deep understanding of how the homework works and is coded.
- Questions will be taken directly from things covered in homework.

Late Policy

You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. You are also responsible for ensuring that what you turned in is what you meant to turn in. Each assignment will have an official due date. Homeworks, only, will be allowed a 24-hour grace period for a 25% penalty. After the grace period absolutely no credit will be given. Therefore, it is your responsibility to plan and ensure that you have backups, early safety submissions, etc.

Academic Misconduct

- ↗ Academic misconduct is taken very seriously in this class.
- ↗ Quizzes, timed labs and the final examination are individual work.
- ↗ Homework assignments may be collaborative, but only at a high level. In addition many homework assignments will be evaluated via demo or code review. During this evaluation, you will be expected to be able to explain every aspect of your submission. Homework assignments will also be examined using electronic computer programs to find evidence of unauthorized collaboration.

Academic Misconduct

- ↗ What is unauthorized collaboration? Each individual programming assignment should be coded by you. You may work with others, sharing ideas and even high-level pseudo-code, but each student must turn in their own version of the assignment.
- ↗ Submissions that are substantially identical will receive a zero and will be forwarded to the Dean of Students' Office of Academic Integrity. Submissions which are copies that have been superficially modified to conceal that they are copies will also be considered unauthorized collaboration.

Academic Misconduct

- You are expressly forbidden to supply a copy of your homework to another student via electronic means. If you supply an electronic copy of your homework to another student and they are charged with copying you will also be charged. This includes storing your code on any site which would allow other parties to obtain your code such as but not limited to public repositories, etc.

Final Exam

**IF YOU ARE LATE OR MISS THE FINAL
EXAM YOU RECEIVE A ZERO**

Final

- The final exam is comprehensive

Need help?

- ↗ Piazza
- ↗ TAs – You may attend any of the TA's office hours
- ↗ Instructor
- ↗ Dean of Students' Office

End of Semester

- There is no time available to review your final at the end of the semester.
- We do not review finals or discuss grades over break.
- You have the entire next semester you are on campus to review your final and all grades and have any problem fixed.

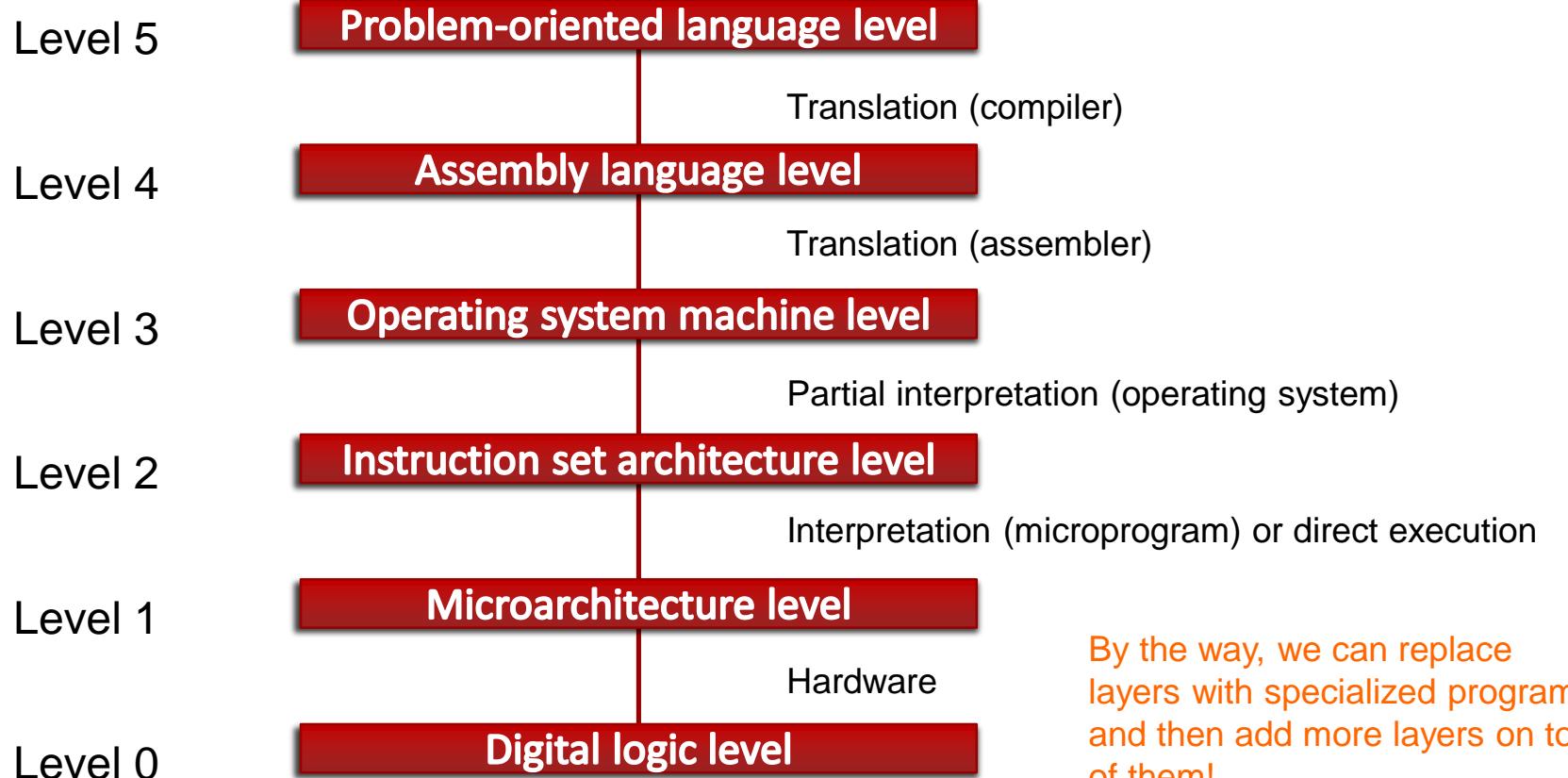


Coming Up: Big Ideas

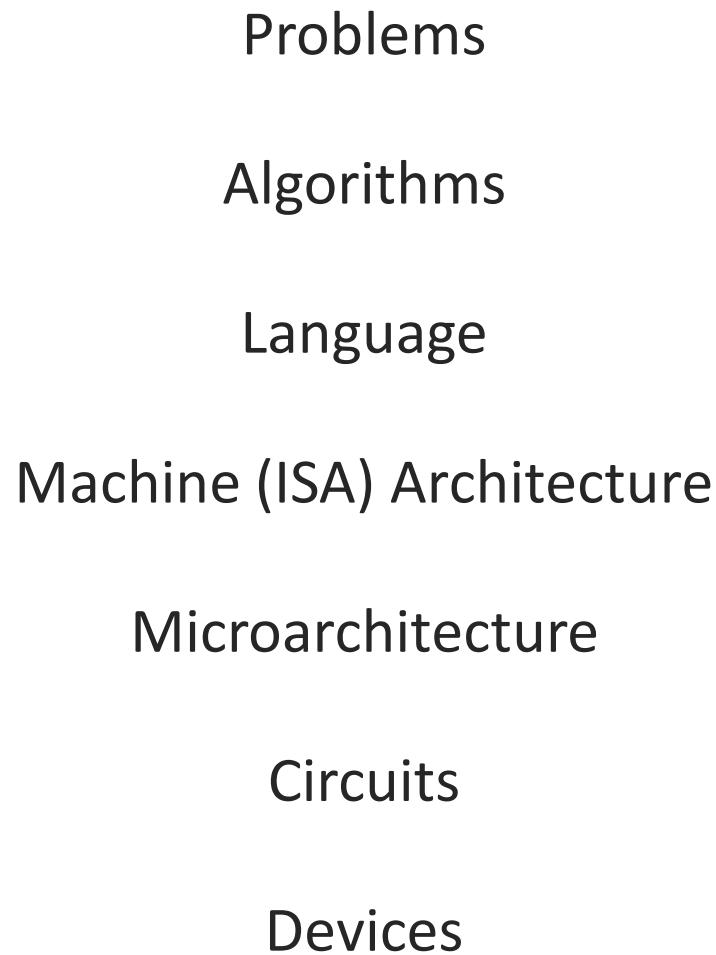
Big Idea #1: All computers can compute the same kinds of things

- We call this Turing-equivalence
- Just about everything that we use for computation can be proved capable of solving the same set of problems.
- That includes Turing machines, stored program computers (and their programming languages), regular expressions, automata theory, formal grammars, etc.

Big Idea #2: Abstraction: Layers Making the Electrons Work



Big Idea #2: Abstraction: Layers Making the Electrons Work



Big Idea #3 Binary

- ↗ Binary is “better” than decimal for electronic computing.
- ↗ Why?
- ↗ Lots of small physical and economic reasons:
 - ↗ It’s easier to determine presence/absence of current rather than magnitude.
 - ↗ Can use lower voltages to distinguish only 0/1 instead of 0/1/2/3/4/5/6/7/8/9, so less power.
 - ↗ Binary-coded decimal math takes more circuitry than pure binary.

Big Idea #4: Computers Store Representations of Something Outside

- ↗ Computers can't store the mathematical abstraction we call a "number". Why?
- ↗ How many digits can a "number" have? How would you build that?
- ↗ So everything in a computer is a finite-sized **representation** of something outside.
- ↗ A bunch of binary digits (bits) is always interpretable as an unsigned whole number. We use that representation often.
- ↗ So we can always claim the bits stored in a computer represent a positive whole number.
- ↗ Is that the end of the story? Definitely not. Stay tuned.

Things to do

- ↗ Get the textbooks.
- ↗ Start reading!
- ↗ Patt, Chapters 1, 2



Questions?



Datatypes I

Outline

- ↗ Binary-Decimal Conversion
- ↗ Arithmetic Operations on Bits
 - ↗ Addition & Subtraction
- ↗ Trouble In Paradise (Don't Be Negative...)
- ↗ Representations & Numbers
- ↗ A New Hope... (2's Complement)
- ↗ Binary-Decimal Conversion (revisited)
- ↗ Sign Extension and Overflow

Representations

- Modern computers effectively store patterns of switches that are set On and Off.
- How do we do work within that limitation?

“When I use a word,’ Humpty Dumpty said in rather a scornful tone, ‘it means just what I choose it to mean — neither more nor less.’

‘The question is,’ said Alice, ‘whether you can make words mean so many different things.’

‘The question is,’ said Humpty Dumpty, ‘which is to be master — that’s all.’”

— Lewis Carroll, Through the Looking Glass

- In that vein, we **choose** representations for numbers, letters, and many other things that are “easy” to work with.

Representations & Numbers

What does this mean to you?

5

The number 5?

An Arabic numeral (digit) that **represents** the **number** 5?

Could there be other representations?

0101₂, V, ...

Representations & Numbers

And what about

$$\sqrt{2}$$

Can you write that number accurately using Arabic numerals?

No? Then how are you going to store it in a computer memory?

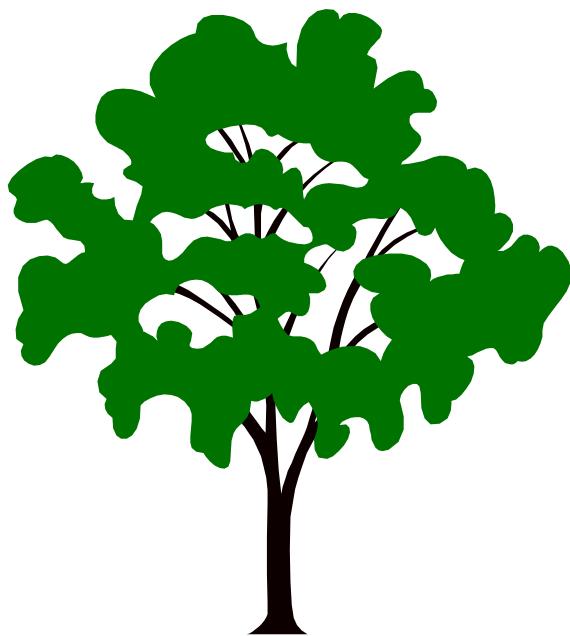
Data Types

- ↗ A **data representation** is the set of values from which a variable, constant, function, or other expression may take its value and includes the meaning of those values. A representation tells the compiler or interpreter how the programmer intends to use it. For example, the process and result of adding two variables differs greatly according to whether they are integers, floating point numbers, or strings.
- ↗ Patt: "We say a particular representation is a **data type** if there are operations in the computer that can operate on information that is encoded in that representation."

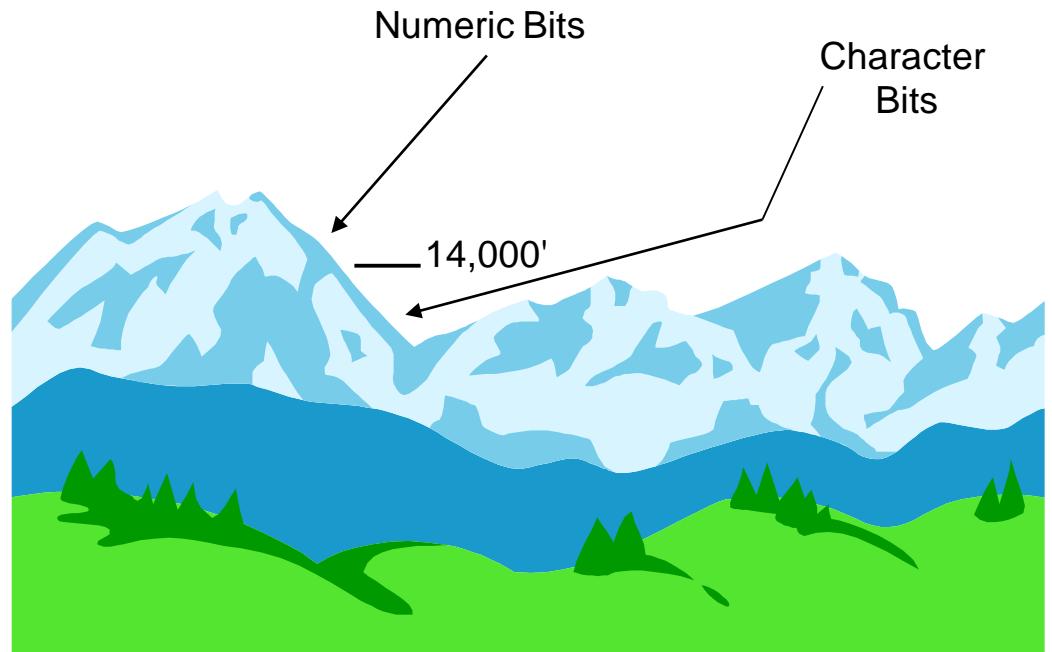
Question

- ↗ What is a bit?
- ↗ “Bit” is short for “binary digit”
- ↗ A bit can take on exactly two values
- ↗ We often refer to those values as 0 and 1

Where do bits come from?



Bitaba Tree



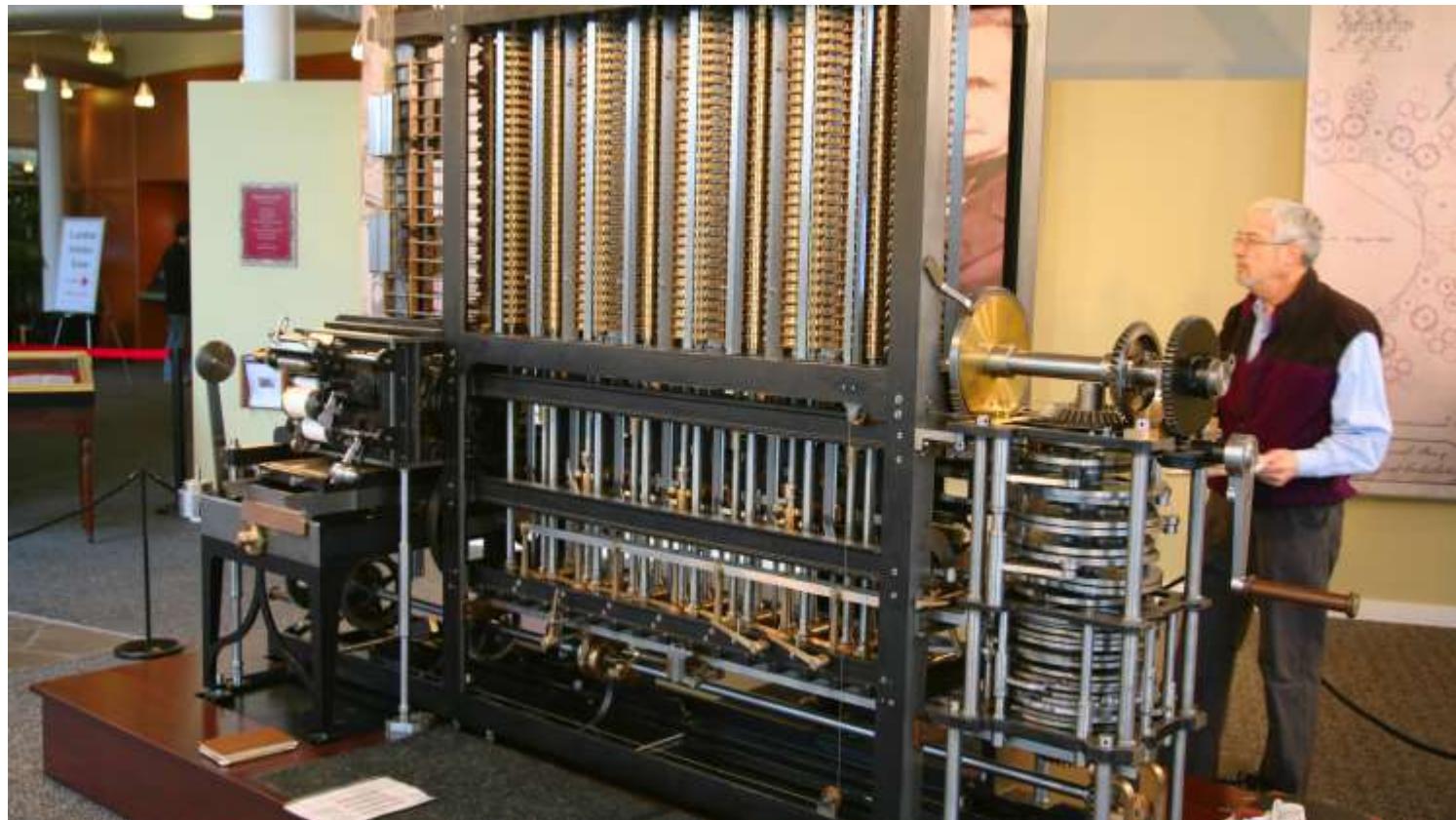
Andes Mountains

NOT!

Question

- The first computers did decimal arithmetic.
Couldn't we just skip binary?

Babbage Difference Engine #2



- Difference Engine #2 Video
- Method of Finite Differences Video
(Math starts at 1:35)

Decimal or Binary?

- ↗ History has pretty well determined that binary is the the most effective way to store and calculate using today's electronics.
- ↗ Only computers in the 40s and 50s attempted to actually store data in decimal.
- ↗ Up through the 70s, computers binary-coded decimal instructions (store in coded binary, calculate in decimal).
- ↗ Decimal storage is very rare in modern computers.

Question

- ↗ What is the smallest number of bits we would need to designate 5 different items or states?
- ↗ How about 16 states?

Unsigned Integers

- ↗ How many different numbers can be represented by 4 bits?
- ↗ How many different numbers can be represented by 7 bits?
- ↗ How many different numbers can be represented by n bits?

Let's Represent Non-Negative Numbers Using Bits

0	0100
1	1001
2	1100
3	0001
4	1011
5	0011
6	1110
7	0101
8	0110
9	0000
10	1101
11	0111
12	1000
13	1111
14	1010
15	0010

Is this a good choice?

How many other choices
are there?

$16! = 20,922,789,888,000$

Maybe we can pick a
better representation?

Perhaps We Should Take Advantage of a Pattern?

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
10	1010
11	1011
12	1100
13	1101
14	1110
15	1111

Quick Review of Base 2 Representation

- There are only 2 values in each place: 0 and 1.
- We assign place-values just like base 10, except they are powers of 2 instead of powers of 10

128	64	32	16	8	4	2	1
0	1	0	0	0	1	0	0

$$0100\ 0100_2 = 68_{10}$$

128	64	32	16	8	4	2	1
0	0	1	1	1	1	1	1

$$0011\ 1111_2 = 63_{10}$$

Memorize Powers of 2

 2^0

1

 2^1

2

 2^9

512

 2^2

4

 2^{10}

1,024

 2^3

8

 2^{11}

2,048

 2^4

16

 2^{12}

4,096

 2^5

32

 2^{13}

8,192

 2^6

64

 2^{14}

16,384

 2^7

128

 2^{15}

32,768

 2^8

256

 2^{16}

65,536

Hmm...

Looks Like Place Value in Binary Arithmetic

0	0 0 0 0	0+0+0+0
1	0 0 0 1	0+0+0+1
2	0 0 1 0	0+0+2+0
3	0 0 1 1	0+0+2+1
4	0 1 0 0	0+4+0+0
5	0 1 0 1	0+4+0+1
6	0 1 1 0	0+4+0+1
7	0 1 1 1	0+4+2+1
8	1 0 0 0	8+0+0+0
9	1 0 0 1	8+0+0+1
10	1 0 1 0	8+0+2+0
11	1 0 1 1	8+0+2+1
12	1 1 0 0	8+4+0+0
13	1 1 0 1	8+4+0+1
14	1 1 1 0	8+4+2+0
15	1 1 1 1	8+4+2+1

Binary to Decimal Conversion

- ↗ Here's a place to practice!
 - ↗ <https://games.penjee.com/binary-numbers-game/>

N = 4	Number Represented
Binary	Unsigned
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	10
1011	11
1100	12
1101	13
1110	14
1111	15

Addition and Subtraction

- Work the same way as base 10. Carry if the bit value is > 1; borrow if less than 0.

$$\begin{array}{r} 0000\ 1111_2 \\ +0001\ 0001_2 \\ \hline \end{array}$$

0010 0000₂

$$\begin{array}{r} 11\ 0000_2 \\ -00\ 0011_2 \\ \hline \end{array}$$

10 1101₂

$$15+17 = 32$$

$$48 - 3 = 45$$

Practice Problems

↗ What is $5 + 2$?

↗ What is $9 + 4$?

↗ What is $10 - 3$?

↗ What is $2 - 6$?

Trouble In Paradise

- ↗ We are used to dealing with positive and negative integers
- ↗ Then we'd better find a way to represent signed integers
- ↗ What are our options?
- ↗ One possible approach: reserve one bit to represent the sign of a number

Signed Integers

- ↗ Signed magnitude
 - ↗ Easy to understand
 - ↗ Fun to be with
 - ↗ Not so good with hardware

N = 4	Number Represented	
Binary	Unsigned	Signed Mag
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-0
1001	9	-1
1010	10	-2
1011	11	-3
1100	12	-4
1101	13	-5
1110	14	-6
1111	15	-7

Signed integers

- ↗ 1's complement
 - ↗ Used in some early computers
 - ↗ To make a negative just flip all the bits

N = 4	Number Represented		
Binary	Unsigned	Signed Mag	1's Comp
0000	0	0	0
0001	1	1	1
0010	2	2	2
0011	3	3	3
0100	4	4	4
0101	5	5	5
0110	6	6	6
0111	7	7	7
1000	8	-0	-7
1001	9	-1	-6
1010	10	-2	-5
1011	11	-3	-4
1100	12	-4	-3
1101	13	-5	-2
1110	14	-6	-1
1111	15	-7	-0

Nagging Concerns...

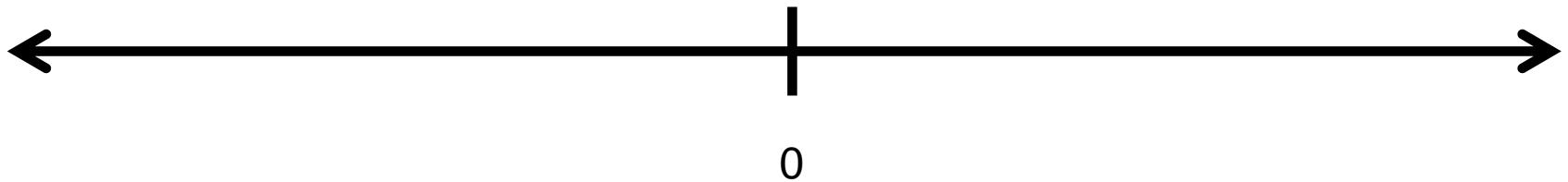
- ↗ Why do I have to think about two different operations: addition and subtraction?
- ↗ How much time will I spend examining the signs and magnitudes of each operand to determine how to find the answer?
- ↗ And what about plus zero and minus zero representing the same number?
- ↗ Is there a better way?...

Reality Sets In

- Any arbitrary scheme of allowing different bit patterns to represent different numbers could be made to work, but there is an especially clever choice...

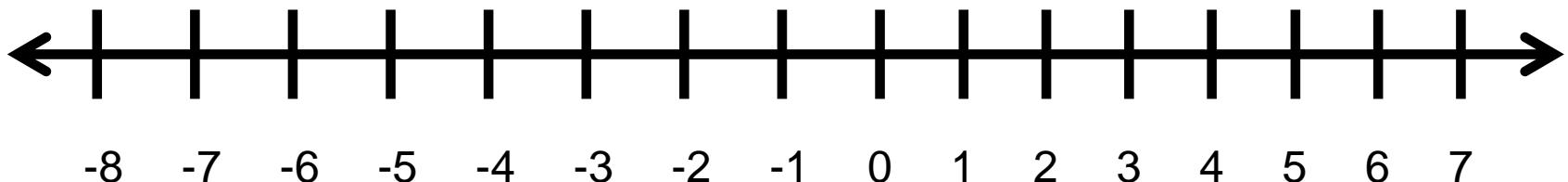
A New Hope...

- ↗ Start with the number line



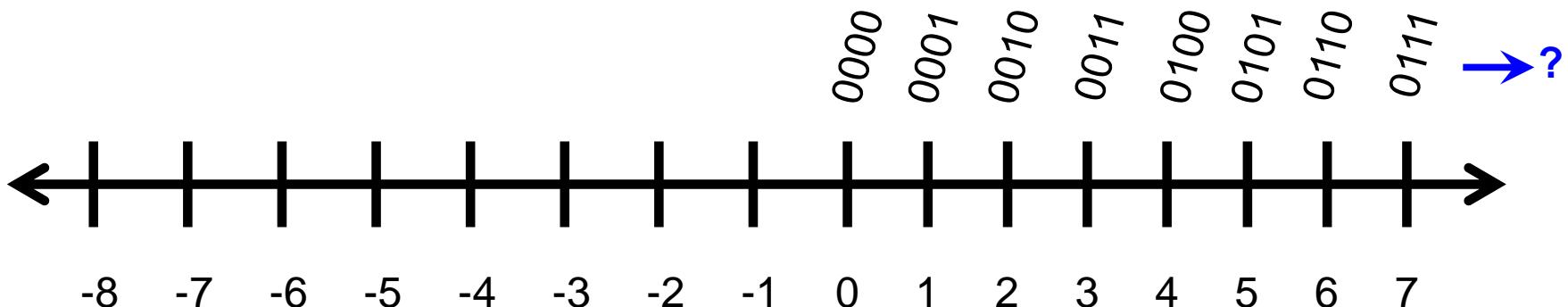
A New Hope...

- ↗ Let's use 4 bits for our example representation
- ↗ This gives us $2^4 = 16$ different values that we can represent
- ↗ Let's split them as evenly as possible between positive and negative values



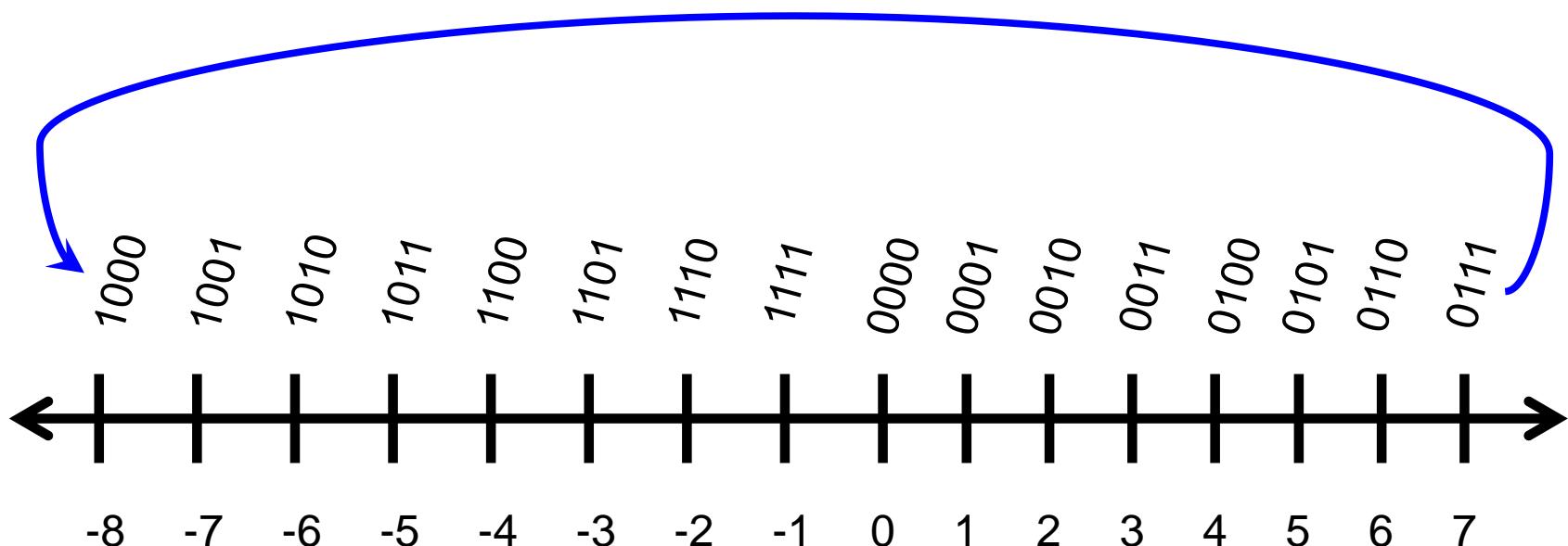
A New Hope...

- ↗ Now, start numbering from 0 using our trusty unsigned binary representation
- ↗ But... what happens when we reach the “end of the line” ?
(hint: think odometer)



A New Hope...

- ↗ Wrap around and keep going!
- ↗ Remember: We're stuck with modular arithmetic – we can't represent all numbers



N = 4	Number Represented				
Binary	Unsigned	Signed Mag	1's Comp	2's Comp	
0000	0	0	0	0	0
0001	1	1	1	1	1
0010	2	2	2	2	2
0011	3	3	3	3	3
0100	4	4	4	4	4
0101	5	5	5	5	5
0110	6	6	6	6	6
0111	7	7	7	7	7
1000	8	-0	-7	-8	
1001	9	-1	-6	-7	
1010	10	-2	-5	-6	
1011	11	-3	-4	-5	
1100	12	-4	-3	-4	
1101	13	-5	-2	-3	
1110	14	-6	-1	-2	
1111	15	-7	-0	-1	

...for Signed integers

- 2's complement
 - Used in almost all computers manufactured today
 - Allows for low cost, high performance circuitry to do math operations
 - There's no need for a hardware subtractor, just a bitwise complement
 - Why does it work?

Useful Properties of 2's Comp

- ↗ There is only one zero
- ↗ Addition still works as expected from binary arithmetic (if you don't wrap)
- ↗ The first bit still indicates the sign (as long as you will tolerate zero being positive)
- ↗ Finding the complement (additive inverse) of a number is easy
- ↗ Since finding the complement is easy, there's no need for subtraction: just complement and add

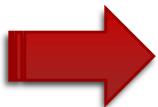
Two's Complement Representation

- ↗ Note there's a useful new pattern:
 - ↗ You can find the two's complement of any number by flipping the bits and adding 1.
 - ↗ And a two's complement conveniently represents the additive inverse of the originally represented number!
- ↗ Repeat the Mantra (negating a number):
 - ↗ Flip the bits
 - ↗ Then add one

Question

How does one compute the additive inverse of a two's complement number?

- A. Take the Boolean NOT of each bit
- B. Take the Boolean NOT of the high-order bit
- C. Take the Boolean NOT of each bit and subtract 1
- D. Take the Boolean NOT of each bit and add 1



Signed Binary to Decimal Conversion

- Given a negative number in two's complement notation how do we convert it to decimal?
 - Flip the bits
 - Then add one
 - Convert to decimal
 - The number is the negative of that decimal num

 1. Consider the four-bit two's complement number: 1011
 2. Flip the bits: 0100
 3. Add 1: 0101
 4. Convert to decimal: +5
 5. So the binary two-s complement 1011 represents -5 in decimal

Which ones to know

- ↗ Be able to do math with:
 - ↗ Unsigned
 - ↗ Two's Complement
- ↗ Only understand the concept of:
 - ↗ Signed magnitude
 - ↗ One's complement
 - ↗ (You don't have to do math with these.)

More Practice Problems

- ↗ What is $5 + 1$?

- ↗ What is $5 - 2$?

- ↗ What is $5 - (-2)$?

- ↗ What is $5 + -5$?

More Practice Problems

5 + 1

$$\begin{array}{r} \textcolor{blue}{0001} \\ 5 \quad 0101 \\ 1 \quad 0001 \\ \hline \text{---} \quad \text{---} \\ 6 \quad 0110 \end{array}$$

5 - 2

Convert to $5 + (-2)$

$$\begin{array}{r} \textcolor{blue}{1100} \\ 5 \quad 0101 \\ -2 \quad 1110 \\ \hline \text{---} \quad \text{---} \\ 3 \quad 0011 \end{array}$$

More Practice Problems

$5 - (-2)$

Convert to $5 + 2$

$$\begin{array}{r} \textcolor{blue}{0000} \\ 5 \quad 0101 \\ 2 \quad 0010 \\ \hline \text{---} \quad \text{---} \\ 7 \quad 0111 \end{array}$$

$5 + (-5)$

$$\begin{array}{r} \textcolor{blue}{1111} \\ 5 \quad 0101 \\ -5 \quad 1011 \\ \hline \text{---} \quad \text{---} \\ 0 \quad 0000 \end{array}$$



Questions?

Question

What is the sum of 101111_2 and 001010_2 ?

(Interpret as unsigned whole numbers.)

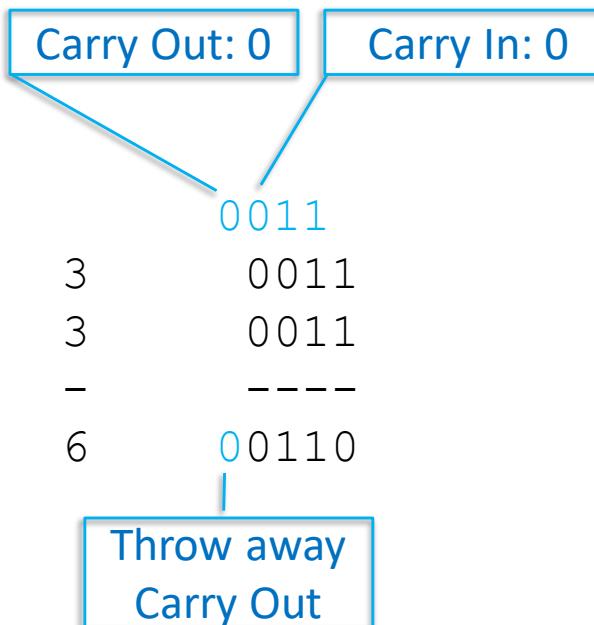
- A. 56
-  B. 111001_2
- C. 100101_2
- D. 47

Overflow (2s comp addition)

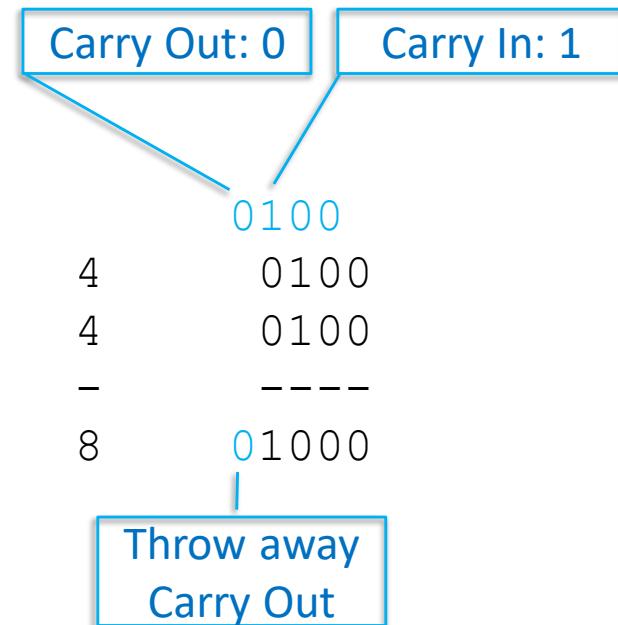
- ↗ Check carry in and out of the leading digit
 - ↗ (most significant digit, on the left)
- ↗ Add two numbers; we have problems if
 - ↗ We get a carry into the sign bit but no carry out
 - ↗ We get a carry out of the sign bit but no carry in
- ↗ If we add a positive and a negative number we won't have a problem
- ↗ Assume 4 bit numbers (-8 : +7) for some examples

Overflow

- If we add two positive numbers and we get a carry into the sign we have a problem



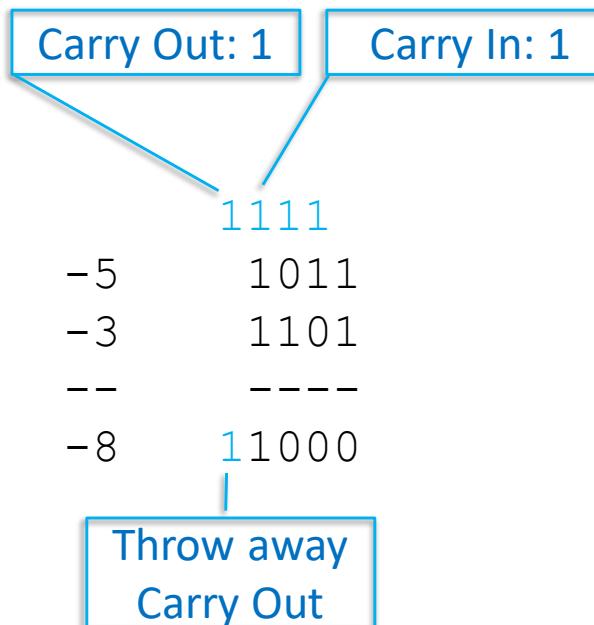
No Overflow



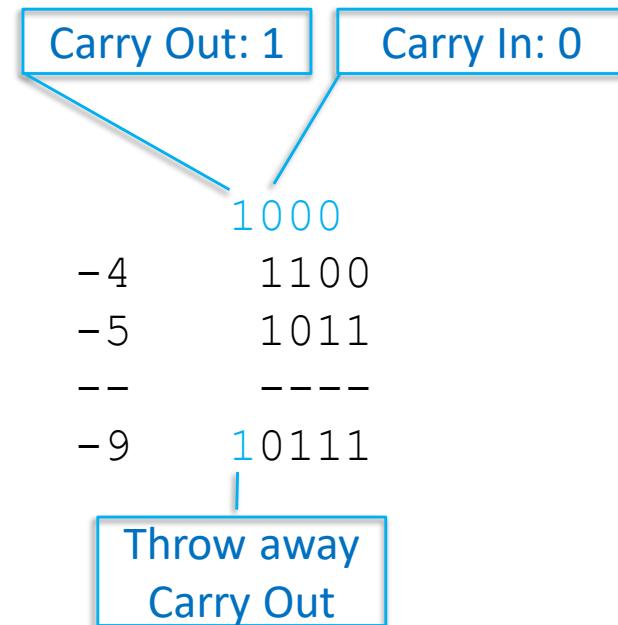
Overflow

Overflow

- If we add two negative numbers and we don't get a carries into and out of the sign bit we have a problem



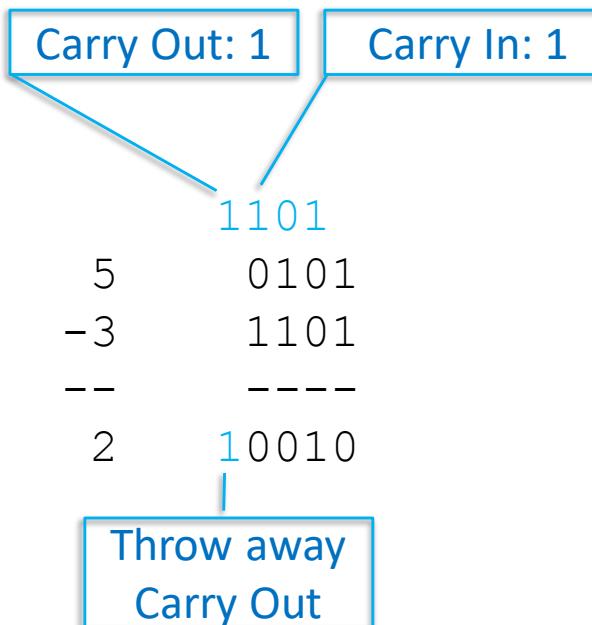
No Overflow



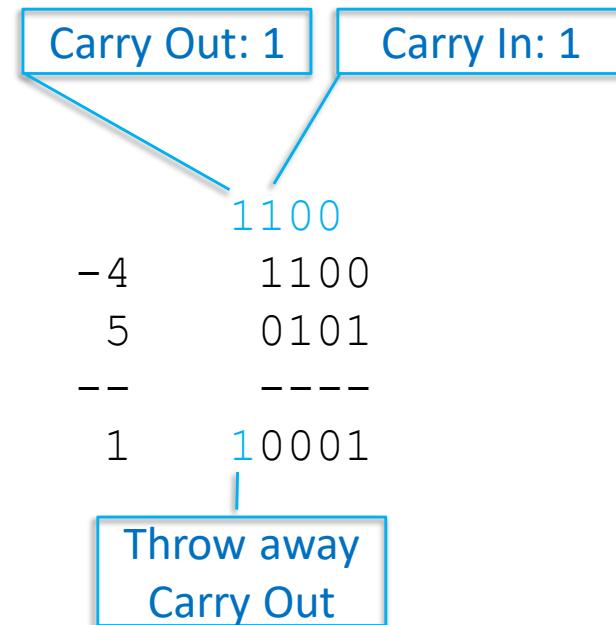
Overflow

Overflow

- If we add a positive and a negative number we won't ever have a problem



No Overflow



No Overflow

Sign Extension

- ↗ Suppose we have a number which is stored in a four bit register
- ↗ We wish to add this number to a number stored in an eight bit register
- ↗ We have a device which will do the addition and it is designed to add two 8 bit numbers
- ↗ What issues do we need to deal with?

Sign Extension

- To add bits to the left of a two's complement number (i.e. make the number of bits bigger while representing the same numeric value)...
- Fill the new bits on the left with the value of the sign bit!
- This is a characteristic of two's complement that you have to remember!

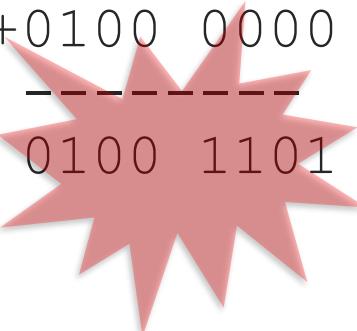
Sign Extension – First Attempt

- Add 3 and 64 (0011 and 01000000)

$$\begin{array}{r} 0011 \\ +0100 \quad 0000 \\ \hline 0100 \quad 0011 \end{array}$$

Correct!

- Add -3 and 64 (1101 and 01000000)

$$\begin{array}{r} 1101 \\ +0100 \quad 0000 \\ \hline 0100 \quad 1101 \end{array}$$


WRONG!

Sign Extension Example

- Add 3 and 64 (0011 and 01000000)

$$\begin{array}{r} 0011 \\ +0100 \quad 0000 \\ \hline 0100 \quad 0011 \end{array}$$

$$\begin{array}{r} 0000 \quad 0011 \\ +0100 \quad 0000 \\ \hline 0100 \quad 0011 \end{array}$$

- Add -3 and 64 (1101 and 01000000)

$$\begin{array}{r} 1101 \\ +0100 \quad 0000 \\ \hline 0100 \quad 1101 \end{array}$$

$$\begin{array}{r} 1111 \quad 1101 \\ +0100 \quad 0000 \\ \hline 10011 \quad 1101 \end{array}$$

Throw away the
carried out 1

What happens when we add a number to itself?

00010000	00011111	00010001	00000101
00010000	00011111	00010001	00000101
-----	-----	-----	-----
00100000	00111110	00100010	00001010

01110011	01010101	00011100	00000001
01110011	01010101	00011100	00000001
-----	-----	-----	-----
11100110	10101010	00111000	00000010

A + A is the same as **2 * A** is the same as **A << 1**

Fractional Binary Numbers

What is the place value of the first digit to the right of a decimal point?

10^{-1}

What is the place value of the first bit to the right of a binary point?

2^{-1}

So, what is the decimal value of the following binary number?

$1.0\ 1$

$= 1.25_{10}$

8	4	2	1	$\wedge .5$.25	.125	.0625
1	0	1	0	1	1	0	0

$1010\wedge1100 = 10.75_{10}$

What base 10 number is represented by

101.011_2

- A. 5.11
- B. 4 5/8
- C. 5 3/8
- D. 5.08





Datatypes II

Outline

- ↗ Logical Operations on Bits
 - ↗ AND, OR, NOT, XOR, (NAND, NOR)
 - ↗ Shift
- ↗ Other Representations
 - ↗ Bit vectors
 - ↗ Hexadecimal
 - ↗ Octal
 - ↗ ASCII
 - ↗ Floating Point

AND

AND	0	1
0		
1		

AND

AND	0	1
0	0	0
1	0	1

Truth Table

A	B	A AND B A & B AB
0	0	
0	1	
1	0	
1	1	

Truth Table

A	B	A AND B A & B AB
0	0	0
0	1	0
1	0	0
1	1	1

What is “Bitwise”?

- ↗ Traditional Boolean functions are defined on Boolean values (i.e. True and False)
- ↗ When we have two strings of bits, we often apply a Boolean function to pairs of respective bits in the two strings
- ↗ We refer to this operation on two arrays of bits as a “bitwise” operation
- ↗ So we might write
$$0110_2 \text{ AND } 0011_2 = 0010_2$$
meaning that we should apply the AND function to each pair of bits

Bitwise AND

0101 AND 0110

(5 & 6)

0101

0110

0100

OR

A	B	A OR B A B A + B
0	0	
0	1	
1	0	
1	1	

OR

A	B	A OR B A B A + B
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise OR

0101 OR 0110

(5 | 6)

0101

0110

0111

NOT

A	NOT A $\sim A$ A'
0	
1	

NOT

A	NOT A
0	1
1	0

Bitwise NOT (Complement)

NOT 0101

~5

0101

1010

XOR

A	B	A XOR B $A \wedge B$
0	0	
0	1	
1	0	
1	1	

XOR

A	B	A XOR B $A \wedge B$
0	0	0
0	1	1
1	0	1
1	1	0

Bitwise XOR

0101 XOR 0110

5^6

0101

0110

0011

NAND

A	B	A NAND B $\sim(A \& B)$
0	0	
0	1	
1	0	
1	1	

NAND

A	B	A NAND B $\sim(A \& B)$
0	0	1
0	1	1
1	0	1
1	1	0

Bitwise NAND

0101 NAND 0110

No C/Java Operator

$\sim(5 \And 6)$

0101

0110

1011

NOR

A	B	A NOR B $\sim(A \mid B)$
0	0	
0	1	
1	0	
1	1	

NOR

A	B	A NOR B $\sim(A \mid B)$
0	0	1
0	1	0
1	0	0
1	1	0

Bitwise NOR

0101 NOR 0110

No C Operator

$\sim(5 \mid 6)$

0101

0110

1000

How Many?

- ↗ How many two-argument boolean functions do you think there are?
- ↗ How can you prove it?
- ↗ Enumerate them?

All the Boolean Functions?

P	Q	FALSE	P AND Q	$\sim(P \rightarrow Q)$ P AND $\sim Q$	P	$\sim(Q \rightarrow P)$ $\sim P$ AND Q	Q	$P \neq Q$ P XOR Q	P OR Q
0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

P	Q	P NOR Q	$P == Q$ $\sim(P \text{ XOR } Q)$	$\sim Q$	$Q \rightarrow P$ P OR $\sim Q$	$\sim P$	$P \rightarrow Q$ $\sim P$ OR Q	P NAND Q	TRUE
0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1

Left Shift

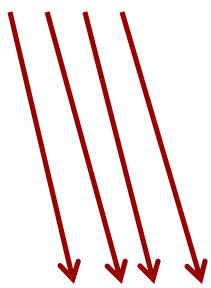
0111 Leftshift 10

$7 \ll 2$



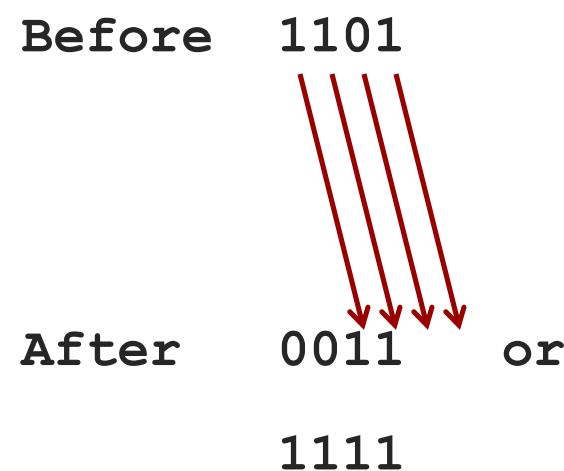
Right Shift

0111 Rightshift 10 7 >> 2

Before 0111

After 0001

Note Well

1101 Rightshift 10 13 >> 2



How would you negate in 2's-complement using
bitwise operators and the plus operator?

How would you negate in 2's complement using
bitwise operators and the plus operator?

$$\sim X + 1$$

Other Representations

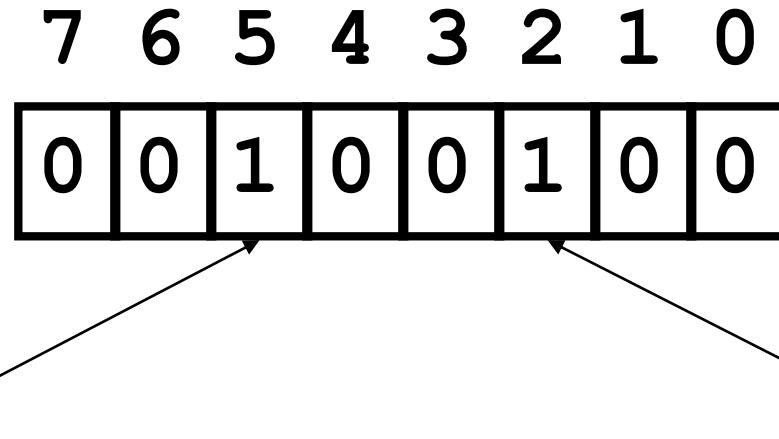


Bit Vectors



Bit Vectors

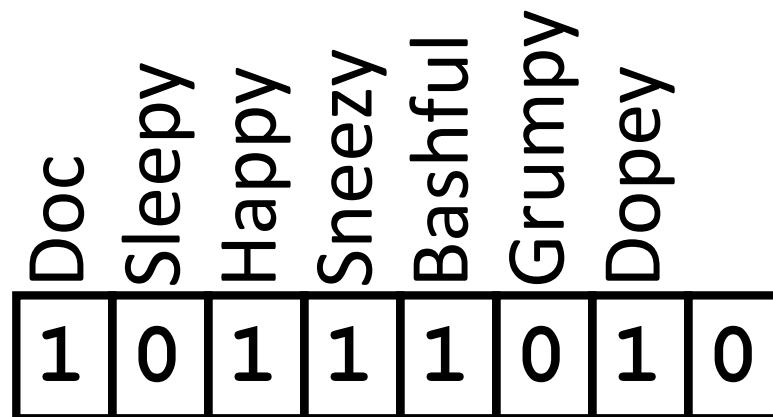
- Sometimes for reasons of space efficiency we can effectively store a group of booleans packed together in a single byte/word/etc.





Bit Vectors

↗ So who's washed their hands!

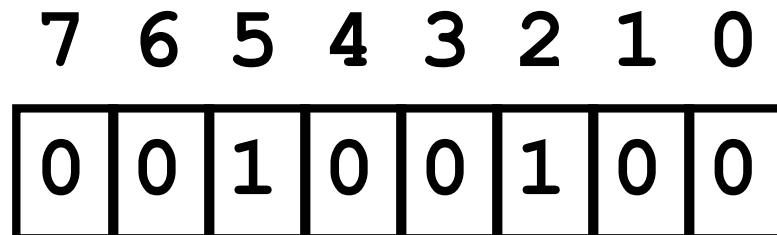


Item 6 is false

Item 3 is false

Bit Vectors

- ↗ How do we manipulate the individual bits in the bit vector?
- ↗ Examples
 - ↗ How do we set bit 6?
 - ↗ How do we clear bit 2?
 - ↗ How do we toggle a bit?
 - ↗ How do we test a bit?



Tallying the Dwarfs

- ☞ $w = 0$ // all bits cleared to 0
 - $w = w \mid 0b10000000$ // Doc
 - $w = w \mid 0b00100000$ // Happy
 - $w = w \mid 0b00010000$ // Sneezy
 - $w = w \mid 0b00001000$ // Bashful
 - $w = w \mid 0b00000010$ // Dopey
- ☞ ***Alternately,***

$w = \sim 0$ // all bits set to 1

$w = w \& 0b10111111$ // Sleepy

$w = w \& 0b11111011$ // Grumpy

7 6 5 4 3 2 1 0

1	0	1	1	1	0	1	0
---	---	---	---	---	---	---	---

0b
means
base 2

Manipulating Bits

- ↗ Often we use a constant (a.k.a. **mask**) with a boolean function (four bit examples)
- ↗ CLEAR :: Identity: $wxyz_2 \& 1111_2 == wxyz_2$
 - ↗ So put a zero in any bit you want to clear
 - ↗ $wxyz_2 \& 1101_2 == wx0z_2$
- ↗ SET :: Identity: $wxyz_2 | 0000_2 == wxyz_2$
 - ↗ So put a one in any bit you want to set
 - ↗ $wxyz_2 | 0100_2 == w1yz_2$
- ↗ TOGGLE: $wxyz_2 ^ 1111_2 = w'x'y'z'_2$
 - ↗ So put a one in any bit you want to toggle
 - ↗ $wxyz_2 ^ 1000_2 == w'xyz_2$

More Manipulating Bits

- ↗ To test a bit, clear all the rest
 - ↗ $wxyz_2 \& 0010_2 == 00y0_2$
 - ↗ Now you can test $00y0_2 == 0000_2$
- ↗ To put a 1 in any bit position n in a mask, shift left by n
 - ↗ $1 << 2 == 0100_2$
- ↗ To put a zero in position in a mask, put a one in that position and complement
 - ↗ $\sim(1 << 2) == 1011_2$
 - ↗ (creates as many leading ones as you need)

Question

We're using 32-bit unsigned binary representation for integers; what value would result from

$$12 = 1100_2 \quad (12 | 7) \wedge 63$$

$$7 = 111_2$$

$$63 = 111111_2 \quad A. \quad 15$$

$$\begin{aligned} 1100_2 | 111_2 \\ = 1111_2 \end{aligned}$$

$$\begin{aligned} 1111_2 \wedge 111111_2 \\ = 110000_2 \quad B. \quad 48 \quad \leftarrow \\ C. \quad 31 \\ D. \quad 0 \end{aligned}$$

$$110000_2 = 48$$

Today's Number: 16,384

You are given a 16-bit unsigned binary number, x . To test if bit 8 (numbered from right to left starting at 0) is 1, you could use

$$x \& 0000000100000000_2 \neq 0$$

- A. $\sim x + 256 == \sim 1$
- B. $x | 256 \neq 0$
- C. $x \& (1<<8) \neq 0$ 
- D. $X \& (1>>8) \neq 0$



Hexadecimal & Octal

Difficult

- Using binary numbers is both a blessing a curse!
 - One can examine directly any particular bit
 - Reading, writing, etc. prone to error

Solution

- ↗ Turns out that mathematics has an answer for us
- ↗ Group bits and assign a single digit to represent each group
- ↗ How big should each group be?

Hint: Use a Base That's a Power of 2!

➤ Base 2^3 , a.k.a. Base 8, a.k.a. Octal!

Base 2	000	111	010	100	101
Base 8	0	7	2	4	5

- $00111010100101_2 = 07245_8$
- 14 bits in 5 octal digits!
- And it works backwards, too!

Base 8	6	4	3	0	2
Base 2	110	100	011	000	010

Use a Base That's a Power of 2!

- ↗ Base 2^4 , aka Base 16, aka Hexadecimal!
 - ↗ Use the digits 0 – 15 and group bits in 4s
 - ↗ Oops! Digits 10-15! We'll just use A-F.

Base 2	1000	1111	0011	1100	0001
Base 16	8	F	3	C	1

- ↗ $10001111001111000001_2 = 8F3C1_{16}$
- ↗ And it too works backwards:

Base 16	F	0	0	D	5
Base 2	1111	0000	0000	1101	0101

In Java (and C)

- Constant integers
 - 456 is decimal
 - 0456 is octal
 - 0x456 is hexadecimal
- 0b010101110 is sometimes used for binary,
but is not standard in Java and C
- This notation often shows up instead of
typographical subscripts!

ASCII



Why
two
codes?

Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char	Dec	Hx	Oct	Char
0	0	000	NUL (null)	32	20	040	SPACE	64	40	100	Ø	96	60	140	`
1	1	001	SOH (start of heading)	33	21	041	!	65	41	101	À	97	61	141	a
2	2	002	STX (start of text)	34	22	042	"	66	42	102	฿	98	62	142	b
3	3	003	ETX (end of text)	35	23	043	#	67	43	103	Ҫ	99	63	143	c
4	4	004	EOT (end of transmission)	36	24	044	\$	68	44	104	҂	100	64	144	d
5	5	005	ENQ (enquiry)	37	25	045	%	69	45	105	҃	101	65	145	e
6	6	006	ACK (acknowledge)	38	26	046	&	70	46	106	҄	102	66	146	f
7	7	007	BEL (bell)	39	27	047	'	71	47	107	҅	103	67	147	g
8	8	010	BS (backspace)	40	28	050	(72	48	110	Ҥ	104	68	150	h
9	9	011	TAB (horizontal tab)	41	29	051)	73	49	111	Ӥ	105	69	151	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	74	4A	112	ڶ	106	6A	152	j
11	B	013	VT (vertical tab)	43	2B	053	+	75	4B	113	ܽ	107	6B	153	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	76	4C	114	ܾ	108	6C	154	l
13	D	015	CR (carriage return)	45	2D	055	-	77	4D	115	ܷ	109	6D	155	m
14	E	016	SO (shift out)	46	2E	056	.	78	4E	116	ܸ	110	6E	156	n
15	F	017	SI (shift in)	47	2F	057	/	79	4F	117	ܹ	111	6F	157	o
16	10	020	DLE (data link escape)	48	30	060	0	80	50	120	ܻ	112	70	160	p
17	11	021	DC1 (device control 1)	49	31	061	1	81	51	121	ܼ	113	71	161	q
18	12	022	DC2 (device control 2)	50	32	062	2	82	52	122	ܾ	114	72	162	r
19	13	023	DC3 (device control 3)	51	33	063	3	83	53	123	ܷ	115	73	163	s
20	14	024	DC4 (device control 4)	52	34	064	4	84	54	124	ܸ	116	74	164	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	85	55	125	ܻ	117	75	165	u
22	16	026	SYN (synchronous idle)	54	36	066	6	86	56	126	ܾ	118	76	166	v
23	17	027	ETB (end of trans. block)	55	37	067	7	87	57	127	ܷ	119	77	167	w
24	18	030	CAN (cancel)	56	38	070	8	88	58	130	ܸ	120	78	170	x
25	19	031	EM (end of medium)	57	39	071	9	89	59	131	ܹ	121	79	171	y
26	1A	032	SUB (substitute)	58	3A	072	:	90	5A	132	ܷ	122	7A	172	z
27	1B	033	ESC (escape)	59	3B	073	;	91	5B	133	[123	7B	173	{
28	1C	034	FS (file separator)	60	3C	074	<	92	5C	134	\	124	7C	174	
29	1D	035	GS (group separator)	61	3D	075	=	93	5D	135]	125	7D	175	}
30	1E	036	RS (record separator)	62	3E	076	>	94	5E	136	^	126	7E	176	~
31	1F	037	US (unit separator)	63	3F	077	?	95	5F	137	_	127	7F	177	DEL

What about these?

Why
is
this
at
the
end?

Fun ASCII Facts

'A' = 65 = 41_{16} = 0100 0001

'a' = 97 = 61_{16} = 0110 0001

'0' = 48 = 30_{16} = 0011 0000

'1' = 49 = 31_{16} = 0011 0001

'2' = 50 = 32_{16} = 0011 0010

...

'9' = 57 = 39_{16} = 0011 1001

Fun ASCII Facts

'A' = 65 = 41_{16} = 0100 0001

'a' = 97 = 61_{16} = 0110 0001

'A' + 32 = 61_{16} = 0110 0001 = 'a'

'B' + 32 = 62_{16} = 0110 0010 = 'b'

'z' - 32 = $5A_{16}$ = 0101 1010 = 'Z'

'5' - '0' = 5 = 0000 0101 = 5

5 + '0' = 35_{16} = 0011 0101 = '5'

Fun ASCII Facts

'A' = 65 = 41₁₆ = 0100 0001

Ctrl-A = SOH = 1 = 1₁₆ = 0000 0001

...

'J' = 74 = 4A₁₆ = 0100 1010

Ctrl-J = LF = 10 = A₁₆ = 0000 1010

...

'M' = 77 = 4D₁₆ = 0100 1101

Ctrl-M = CR = 13 = D₁₆ = 0000 1101

Question

Which of these also represents the value 717742_8 ?

$$\begin{aligned} 717742_8 \\ = 111\ 001 \\ \quad 111\ 111 \\ \quad 100\ 010 \\ = \textcolor{red}{111\ 001} \\ \textcolor{red}{111\ 111} \\ \textcolor{green}{100\ 010} \\ = 0011 \\ \quad 1001 \\ \quad 1111 \\ \quad 1110 \\ \quad 0010 \\ = 39FE2_{16} \end{aligned}$$

A. $39FE2_{16}$



B. $E7F88_{16}$

C. 717742_{16}

D. $8D5_{16}$

Question

'P' = 0x50

$$\begin{aligned}0x50 + 32 \\= 0x70\end{aligned}$$

$$\begin{aligned}0x50 + 0x20 \\= 0x70\end{aligned}$$

$$\begin{aligned}0x50 | 0x20 \\= 0x70\end{aligned}$$

$$0x70 = 'p'$$

If you are given an 8-bit number containing an ASCII representation of the letter 'P', which expression below will compute the ASCII representation of 'p'.

- A. 'P' + 32
- B. 'P' + 0x20
- C. 'P' | 0b00100000
- D. All of the above





Floating Point

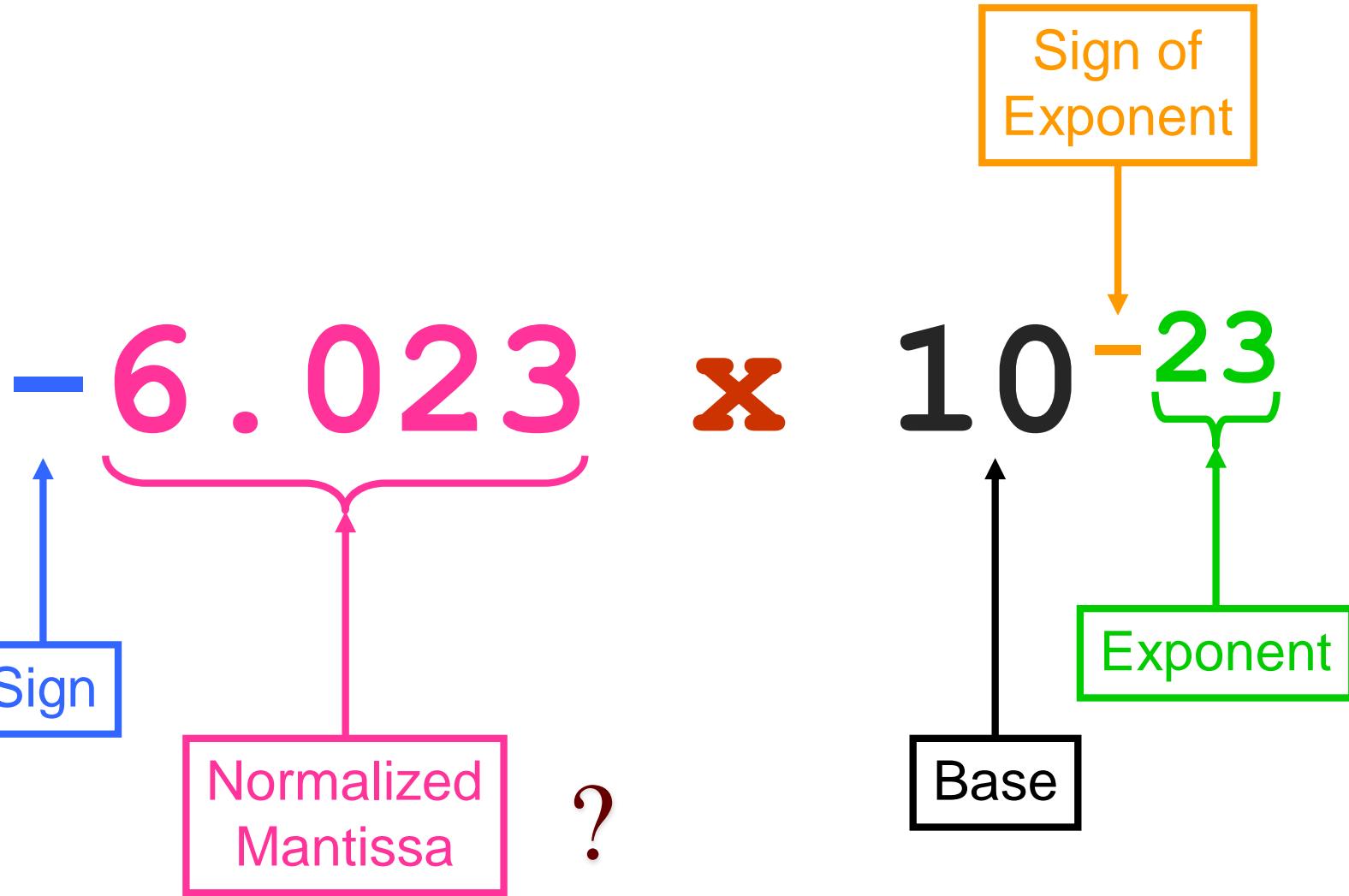
Transcendental Numbers

- ↗ In 1897, a bill was proposed to the Indiana Legislature that set the value of π to exactly 3.2
- ↗ (To their credit they didn't consider it for long; it never came to a vote)
- ↗ Can computers represent π correctly?
- ↗ Can we at least do better than 3.2?

Historically

- ↗ Initially hardware manufacturers used whatever they thought was appropriate given their market and/or technology required.
- ↗ IBM, DEC, CDC, Burroughs, Univac, NCR, Honeywell, GE, RCA, etc. each had their own formats and in fact multiple formats
- ↗ Typical implementations might range from 32 up to 128 bits. Common to find multiple formats available (i.e. float and double)
- ↗ 1985 IEEE published Floating Point Standard
 - ↗ *ANSI/IEEE Standard 754-1985, Standard for Binary Floating Point Arithmetic*

Scientific Notation



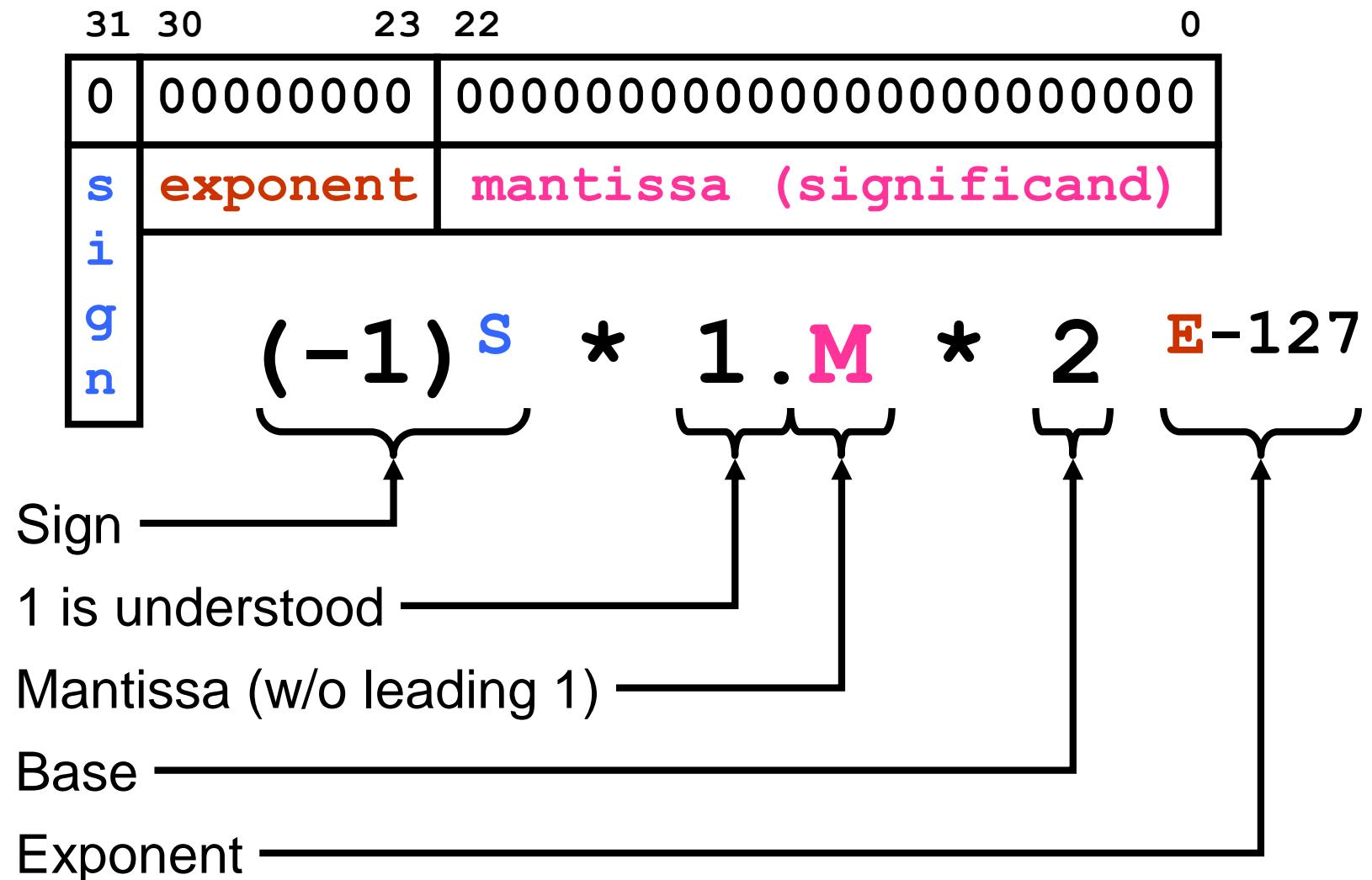


How would you do it?

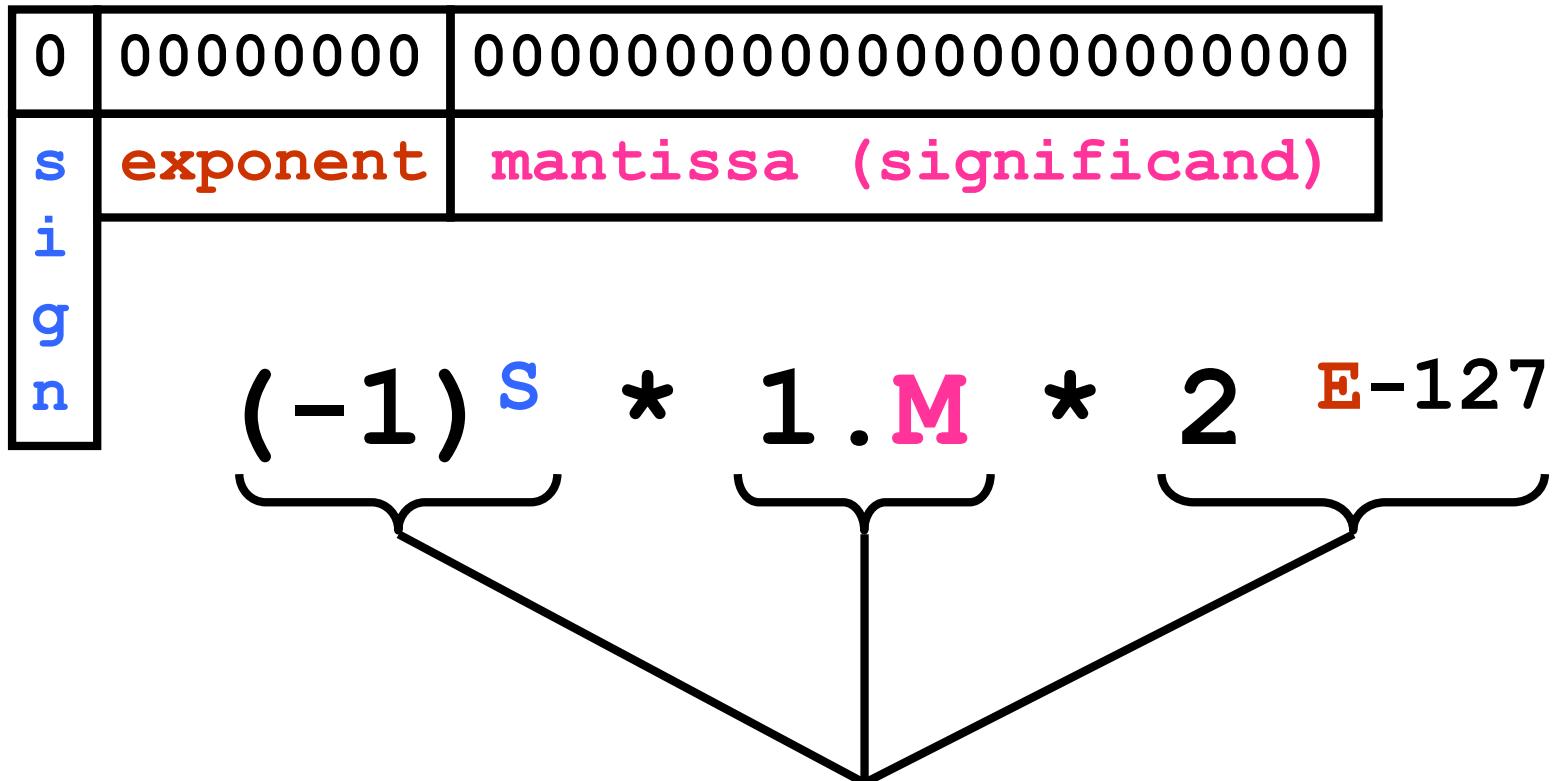
Binary Floating Point Representation

- ↗ Same basic idea as scientific notation
- ↗ Modifications and improvements based on a long history: IEEE-754 standard
 - ↗ Precise representation at the bit level
 - ↗ Precise behavior of arithmetic operations
 - ↗ Efficiency (Space & Time)
 - ↗ Additional requirements
 - ↗ Special values: not-a-number, +/-infinity, etc.
 - ↗ Correct rounding
 - ↗ Sortable without FP hardware

IEEE-754



IEEE-754



Can any of these equal 0?

So how can we represent 0?

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i g n	(-1) ^S * 1.M * 2 ^{E-127}	

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Can be written...

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i g n	(-1) ^S * 2 ^{E-127} * 1.M	

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 00000000 000000000000000000000000 = 0

1 00000000 000000000000000000000000 = -0

0	00000000	000000000000000000000000
s	exponent	mantissa (significand)
i		
g		
n		

$(-1)^S \times 2^{E-127} \times 1.M$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number



Perhaps Some Automation?

<https://www.h-schmidt.net/FloatConverter/IEEE754.html>

0	11111111	00000000000000000000000000000000	= Infinity
1	11111111	00000000000000000000000000000000	= -Infinity

0	00000000	00000000000000000000000000000000	
s	exponent	mantissa (significand)	
i			
g n	(-1) ^S	\times	$2^{E-127} \times 1.M$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 11111111 000001000000000000000000 = NaN

1 11111111 001000100010010101010101 = NaN

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i g n	$(-1)^S \times 2^{E-127} \times 1.M$	

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Not a Number (NaN)

- ↗ Suppose A is a floating point number set to NaN
- ↗ A != B is *true*, when B is another floating point number, NaN, infinity, or anything
- ↗ Interestingly A != A is *true* as well
 - ↗ Not equal to itself
- ↗ Also, if A or B is NaN, the following are always *false*:
 $A < B, A > B, A == B$

0 10000000 000000000000000000000000 = +1 * $2^{(128-127)}$ * 1.0 = 2

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i g n	$(-1)^S * 2^{E-127} * 1.M$	

	$E == 0$	$0 < E < 255$	$E == 255$
$M == 0$	0	Powers of Two	∞
$M \neq 0$	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 10000001 10100000000000000000000000000000 = +1 * $2^{(129-127)}$ * 1.101 = 6.5

1 10000001 10100000000000000000000000000000 = -1 * $2^{(129-127)}$ * 1.101 = -6.5

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i		
(-1) ^S * 2 ^{E-127} * 1.M		

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

Edge case for tiny numbers: E==0 is special, so 2^{-126} is smallest exponent; we use underflow case for anything smaller, like 2^{-127} ; note E==0 is computed as E==1

$$0 \ 00000001 \ 000000000000000000000000 = +1 * 2^{(1-127)} * 1.0 = 2^{(-126)}$$

$$0 \ 00000000 \ 100000000000000000000000 = +1 * 2^{(-126)} * 0.1 = 2^{(-127)}$$

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i		
g		
n		$(-1)^S * 2^{E-127} * 1.M$

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

0 00000000 00000000000000000000000000000001

$$= +1 * 2^{(-126)} * 0.00000000000000000000000000000001$$

$$= 2^{(-149)} \text{ (Smallest positive value)}$$

0	00000000	00000000000000000000000000000000
s	exponent	mantissa (significand)
i g n	$(-1)^S * 2^{E-127} * 1.M$	

	E == 0	0 < E < 255	E == 255
M==0	0	Powers of Two	∞
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

$$\begin{aligned}E &= \\10001110_2 &= \\128+8+4+2 &= \\&= 142\end{aligned}$$

$$\begin{aligned}142 - 127 &= \\&= 15\end{aligned}$$

$$\begin{aligned}M &= 0 \\1_2 \text{ followed by } 23 \text{ zeros} &= \\&= 1\end{aligned}$$

$$\begin{aligned}1 * 2^{15} &= \\&= 32768\end{aligned}$$

What number is represented by this 32-bit IEEE-754 encoding?

0 10001110 00000000000000000000000000000000

- A. 0
- B. 32768 
- C. +Infinity
- D. 2^{142}

	E == 0	0 < E < 255	E == 255
M == 0	0	Powers of Two	∞
M != 0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number

```

0 11111111 000001000000000000000000 = NaN
1 11111111 0010001000100101010101010 = NaN

0 11111111 000000000000000000000000 = Infinity
0 10000001 101000000000000000000000000 = +1 * 2^(129-127) * 1.101 = 6.5
0 10000000 000000000000000000000000 = +1 * 2^(128-127) * 1.0 = 2
0 00000001 000000000000000000000000 = +1 * 2^(1-127) * 1.0 = 2^(-126)
0 00000000 10000000000000000000000000 = +1 * 2^(-126) * 0.1 = 2^(-127)
0 00000000 0000000000000000000000001
    = +1 * 2^(-126) * 0.0000000000000000000000001
    = 2^(-149) (Smallest positive value)
0 00000000 000000000000000000000000 = 0
1 00000000 000000000000000000000000 = -0
1 10000001 1010000000000000000000000 = -1 * 2^(129-127) * 1.101 = -6.5
1 11111111 000000000000000000000000 = -Infinity

```

Conversion

- In CS 1331 we noted:

```
float f;  
  
int i;  
  
f = i;  
  
i = (int)f;
```

- What does this imply?
 1. Converting floats to ints we may lose information
 2. Converting ints to floats we may lose information
 3. Converting either way we may lose information

What's Up Here!

```
#include <stdio.h>

int main()
{
    float f;
    int i = 1234567897;
    int j;
    f = i;
    j = (int)f;
    printf("i = %d  f = %f  j = %d\n", i, f, j);
    return 0;
}

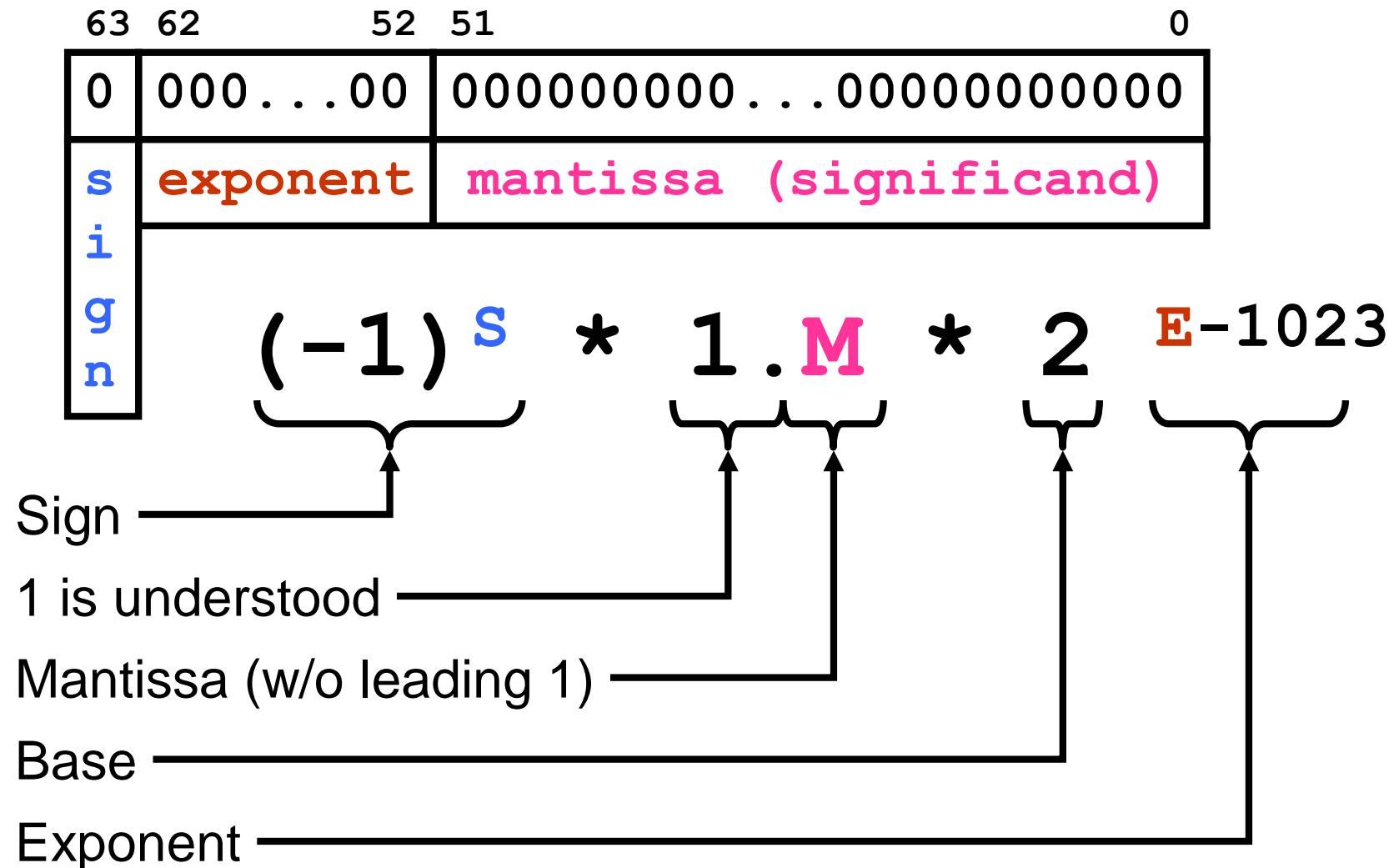
$ ./demo
i = 1234567897  f = 1234567936.000000  j = 1234567936
```

Reality

Int: 00000000000000000000000000000000

Float: 0 00000000 000000000000000000000000

IEEE-754 Double



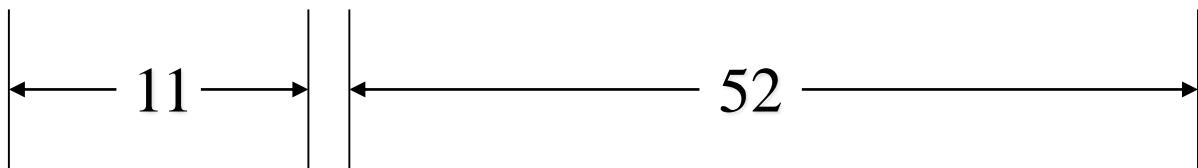
Specials

63 62		52 51		0
0	00000000	00000000000000000000000000000000		
s i g n	exponent	mantissa (significand)		
	(-1) ^S	*	1.M	* 2 ^{E-1023}
	E == 0	0 < E < 2047	E == 2047	
M==0	0	Powers of Two	∞	
M!=0	Non-normalized typically underflow	Ordinary Old Numbers	Not A Number	

Double (64 bits)

Int: 00000000000000000000000000000000

Double: 0 000...00 000000000...000000000



Better?

```
#include <stdio.h>
```

```
int main()
{
    double d;
    int i = 1234567897;
    int j;
    d = i;
    j = (int)d;
    printf("i = %d  d = %f  j = %d\n", i, d, j);
    return 0;
}

$ ./demo
i = 1234567897  d = 1234567897.000000  j = 1234567897
```

Comparing FP Numbers

- ↗ The layout of the representation allows certain operations (like $>$) to be performed with no conversion
- ↗ Compare:
 - $3.67 \times 10^{14} = 0x57a841ab$
 - $2.89 \times 10^{16} = 0x5acd58cb$
- ↗ Notice the exponent bits come first in the representation, so an integer comparison can work if the numbers are the same sign

Comparing Two FP Numbers

- ↗ If either is NaN, the comparison is defined as “unordered” (*all comparisons to it except != are false*)
- ↗ If either is -0.0, replace with +0.0
- ↗ If the signs (high bit) are different, the positive number is bigger
- ↗ Compare the rest of the bits as integers
- ↗ If the signs are both negative reverse the comparison

FP Comparison

A 0 10111000 11001010 11001010 1100101

B 0 10111000 11101010 11001010 1100101

$|B| > |A|$

- ↗ Treat as 31-bit unsigned whole numbers
 - ↗ Without the sign bits (bit 31)
- ↗ And compare the two magnitudes bit by bit, left to right – until you find different values
- ↗ Both are positive, so B>A

What do you need to know?

- ↗ Given the 6-case chart of FP numbers, be able to
 - ↗ Know what each case means and recognize encoded FP numbers that fit each case
 - ↗ Know that in the 32-bit form the exponent is biased by 127
 - ↗ From an encoded FP number, be able to show its value in the form of $M * 2^E$
 - ↗ Be able to compare encoded FP numbers for $<$, $>$, $=$, $!=$
- ↗ Understand
 - ↗ Converting decimal FP to IEEE-754 and vice versa
 - ↗ Precision issues in converting between integer and FP representations

Given these four IEEE-754 representations, which are the largest and smallest?

NaN #1: 1 1111111 00100010001001010101010

6.5 #2: 0 10000001 1010000000000000000000000000

2 #3: 0 10000000 0000000000000000000000000000

-Infinity #4: 1 1111111 0000000000000000000000000000

A. #1 is largest, #2 is smallest

B. #2 is largest, #4 is smallest 

C. #3 is largest, #1 is smallest

D. #4 is largest, #2 is smallest

Bitwise Review

- ↗ Can only be applied to integral operands
 - ↗ *that is, char, short, int and long*
 - ↗ **(signed or unsigned)**
 - & **Bitwise AND**
 - | **Bitwise OR**
 - ^ **Bitwise XOR**
 - << **Shift Left**
 - >> **Shift Right**
 - ~ **1's Complement (Inversion)**

Bitwise Questions

1 & 3

1

1 & 2

0

x << -2 Legal?

No!

x << 2 Write it another way?

x * 4

What does right shifting do to signed vars?

depends

x = x & ~077

Clears last six bits of x to zero!

Bitwise Questions

Why is `x = x & ~077` better than
`x = x & 0177700`

Why is $x = x \& \sim 077$ better than
 $x = x \& 0177700$

$x = 1010101010101010$

$077 = 000000000111111$

$\sim 077 = 111111111000000$

$x = 1010101010101010$
 $\sim 077 = 111111111000000$

} &

$1010101010\textcolor{red}{000000}$

Why is `x = x & ~077` better than
`x = x & 0177700`

$$x = 1010101010101010$$

$$077 = 000000000111111$$

$$\begin{array}{rcl}
 x & = & 1010101010101010 \\
 0177700 & = & 1111111111000000 \\
 & & \hline
 & & 1010101010\textcolor{red}{000000}
 \end{array} \quad \left. \right\} \&$$

But what if the word size is bigger than
16 bits?

Why is $x = x \& \sim 077$ better than $x = x \& 0177700$

$x = 11110000111100001010101010101010$

$077 = 000000000000000000000000000011111$

$\sim 077 = 1111111111111111111111111111000000$

$x = 11110000111100001010101010101010$

$\sim 077 = 1111111111111111111111111111000000$

}

$11110000111100001010101010\textcolor{red}{000000}$

Why is $x = x \& \sim 077$ better than $x = x \& 0177700$

$x = 11110000111100001010101010101010$

$0177700 = 000000000000000011111111000000$

$x = 11110000111100001010101010101010$

$0177700 = 000000000000000011111111000000$

$00000000000000001010101010\textcolor{red}{000000}$

Bitwise Questions

What does this do?

```
(x >> (p+1-n)) & ~(~0 << n);
```

What does this do?

(x >> (p+1-n)) & ~(~0 << n);

P=15 , N=3

332222222222211111111111

10987654321098765432109876543210

`x=001010101001001010101010101101101010101`

$$(x \gg (p+1-n)) \implies (x \gg (15+1-3)) \\ (x \gg 13)$$

00000000000000000010101010010010101

~0

$$\sim 0 \ll n \implies \sim 0 \ll 3$$

$\sim (\sim 0 \ll 3)$

Bitwise Questions

Why is $x = x \& \sim 077$ better than
 $x = x \& 0177700$

First one is independent of word length
(no extra cost...evaluated at compile time)

What does this do?

```
(x >> (p+1-n)) & ~(~0 << n);  
/* getbits: get n bits from position p */  
unsigned getbits(unsigned x, int p, int n)  
{  
    return (x >> (p+1-n)) & ~(~0 << n);  
}
```

Questions?

- ↗ Logical Operations on Bits
 - ↗ AND, OR, NOT, XOR, (NAND, NOR)
 - ↗ Shift
- ↗ Other Representations
 - ↗ Bit vectors
 - ↗ Hexadecimal
 - ↗ Octal
 - ↗ ASCII
 - ↗ Floating Point
- ↗ Bitwise review

Digital Logic I



- ↗ Transistors
- ↗ Logic Gates
 - ↗ NOT, OR, NOR, AND, NAND
 - ↗ DeMorgan's Law
 - ↗ Larger Gates
- ↗ Combinational Logic Circuits
 - ↗ Decoder, MUX, Full Adder, PLA,
 - ↗ Logical Completeness
- ↗ Simplification
 - ↗ Boolean
 - ↗ Karnaugh Maps
 - ↗ PLA/PGA

Nomenclature



Open Switch

No current can flow



Closed Switch

Current can flow

Our Story Begins

George Boole



library.thinkquest.org

George Boole was an English mathematician, philosopher and logician. His work was in the fields of differential equations and algebraic logic, and he is now best known as the author of *The Laws of Thought*. [Wikipedia](#)

Born: November 2, 1815, Lincoln

Died: December 8, 1864, Ballintemple, Cork

Spouse: Mary Everest Boole (m. 1855)

Children: Alicia Boole Stott, Ethel Lilian Voynich, Lucy Everest Boole, Mary Ellen Boole Hinton, Margaret Taylor

Awards: Royal Medal

Telegraph

Samuel Morse



en.wikipedia.org

Samuel Finley Breese Morse was an American contributor to the invention of a single-wire telegraph system based on European telegraphs, co-inventor of the Morse code, and an accomplished painter. [Wikipedia](#)

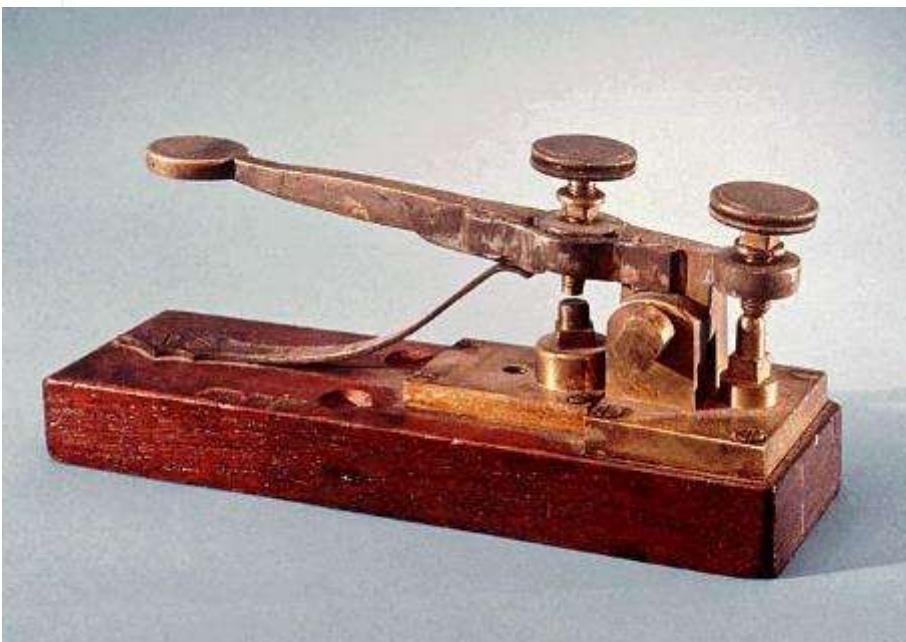
Born: April 27, 1791, Charlestown

Died: April 2, 1872, New York City

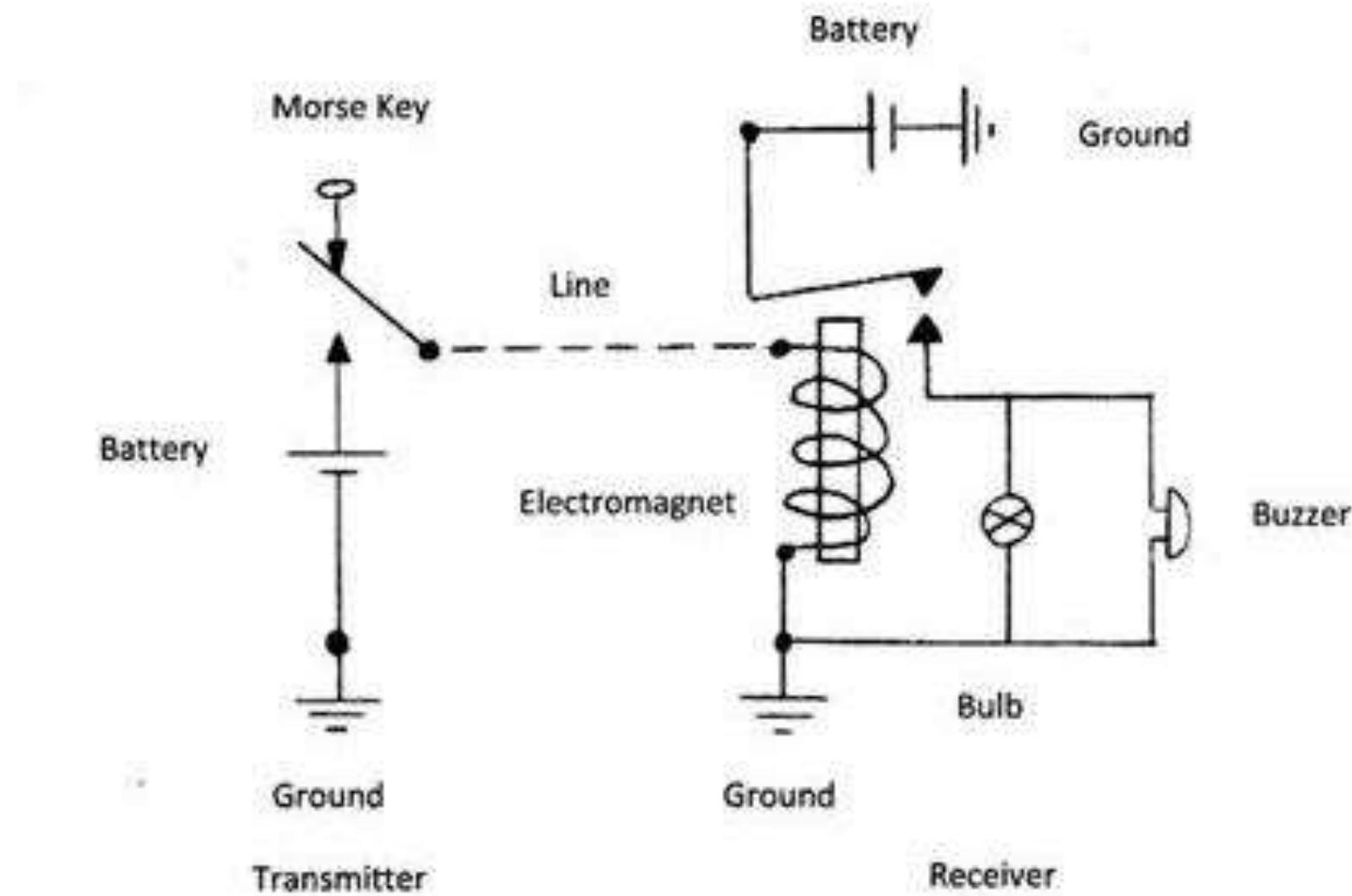
Education: Yale University, Phillips Academy

Parents: Jedidiah Morse, Elizabeth Ann Finley Breese

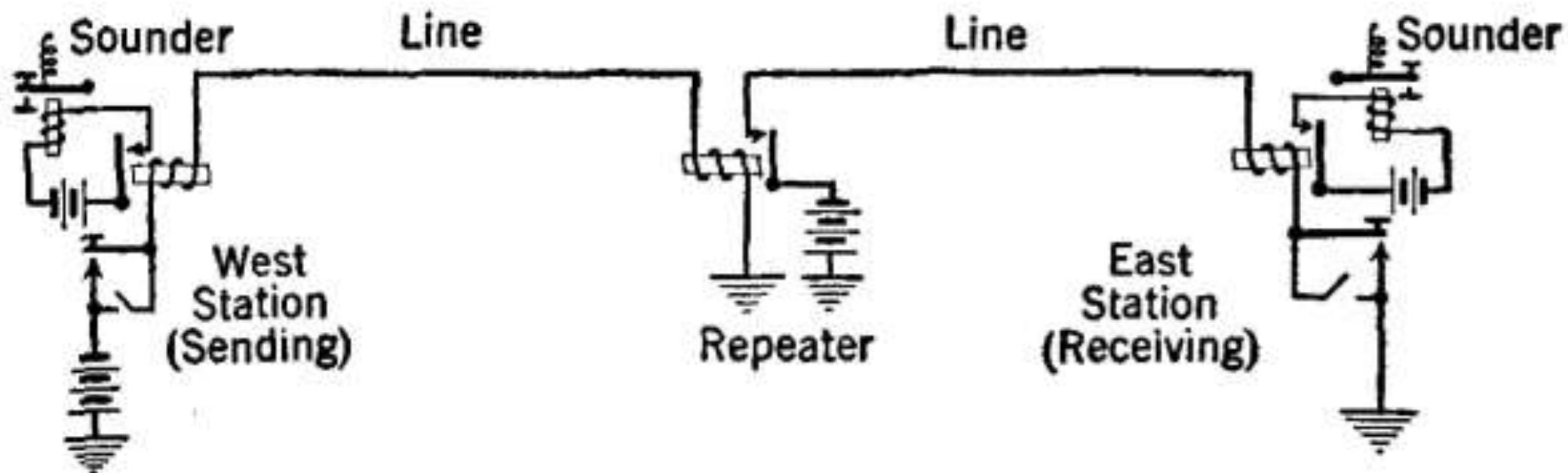
Siblings: Sidney Edwards Morse



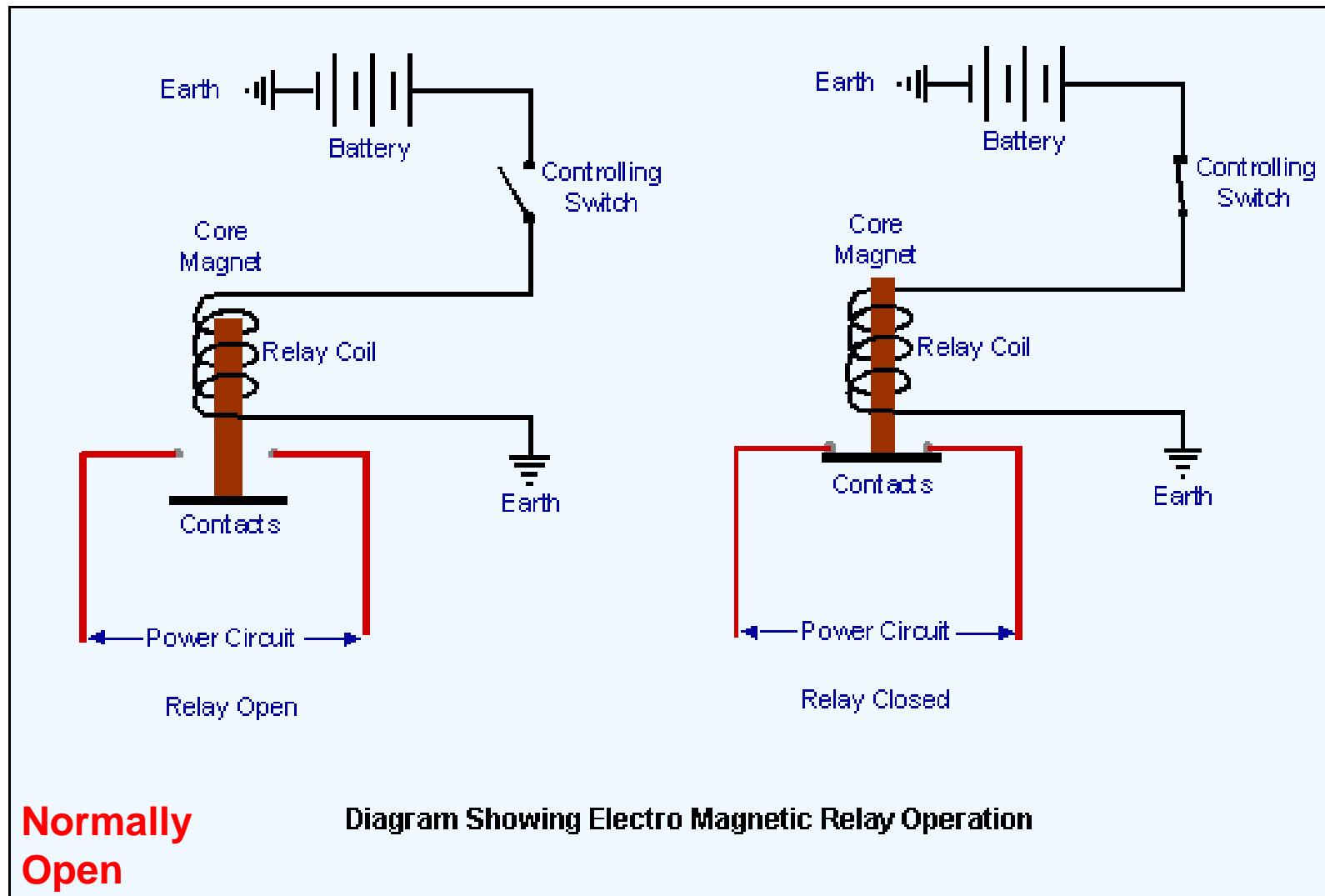
Telegraph Schematic



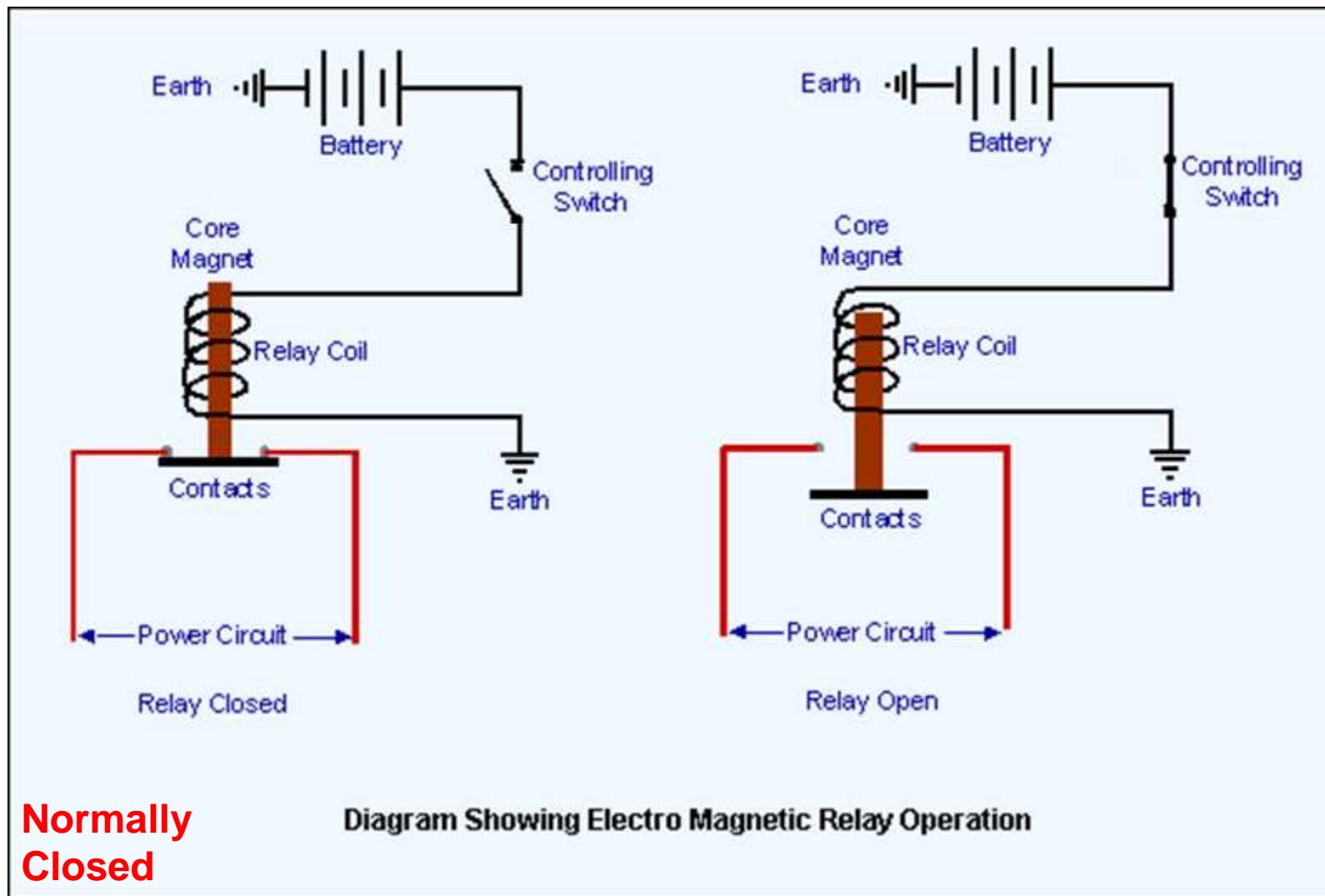
Repeater



Relay



Relay



Claude Shannon

Claude Elwood Shannon



en.wikipedia.org

Claude Elwood Shannon was an American mathematician, electronic engineer, and cryptographer known as "the father of information theory". Shannon is famous for having founded information theory with one landmark paper published in 1948. [Wikipedia](#)

Born: April 30, 1916, Petoskey

Died: February 24, 2001, Medford

Books: [The mathematical theory of communication](#), Claude Elwood Shannon

Education: [Massachusetts Institute of Technology](#), [University of Michigan](#)

Awards: [IEEE Medal of Honor](#), [National Medal of Science for Engineering](#), [More](#)

Claude Shannon's Masters Thesis



A SYMBOLIC ANALYSIS
OF
RELAY AND SWITCHING CIRCUITS

by

Claude Elwood Shannon
B.S., University of Michigan
1936

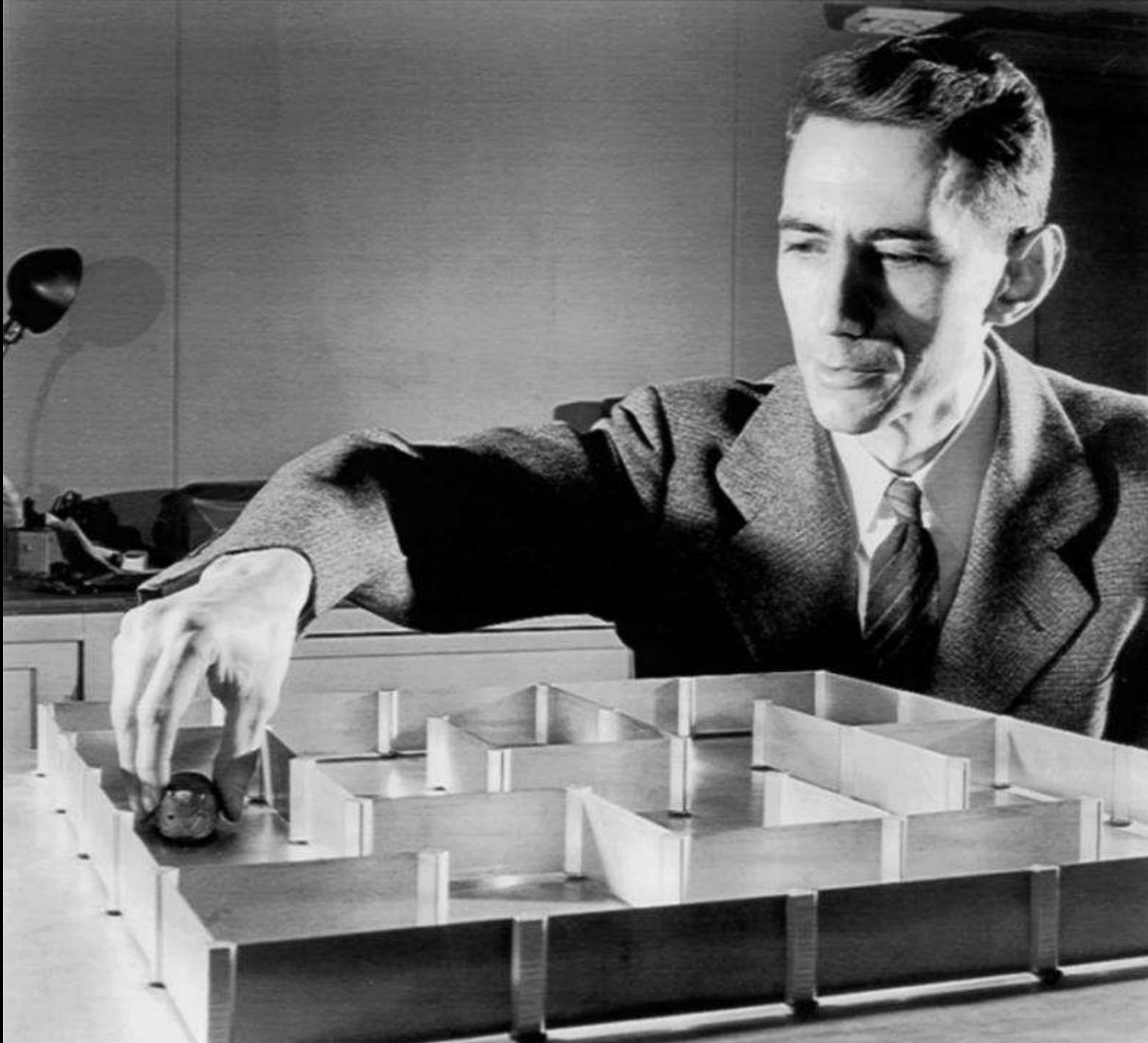
Submitted in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
from the
Massachusetts Institute of Technology
1940

Signature of Author _____

Department of Electrical Engineering, August 10, 1937

Signature of Professor
in Charge of Research _____

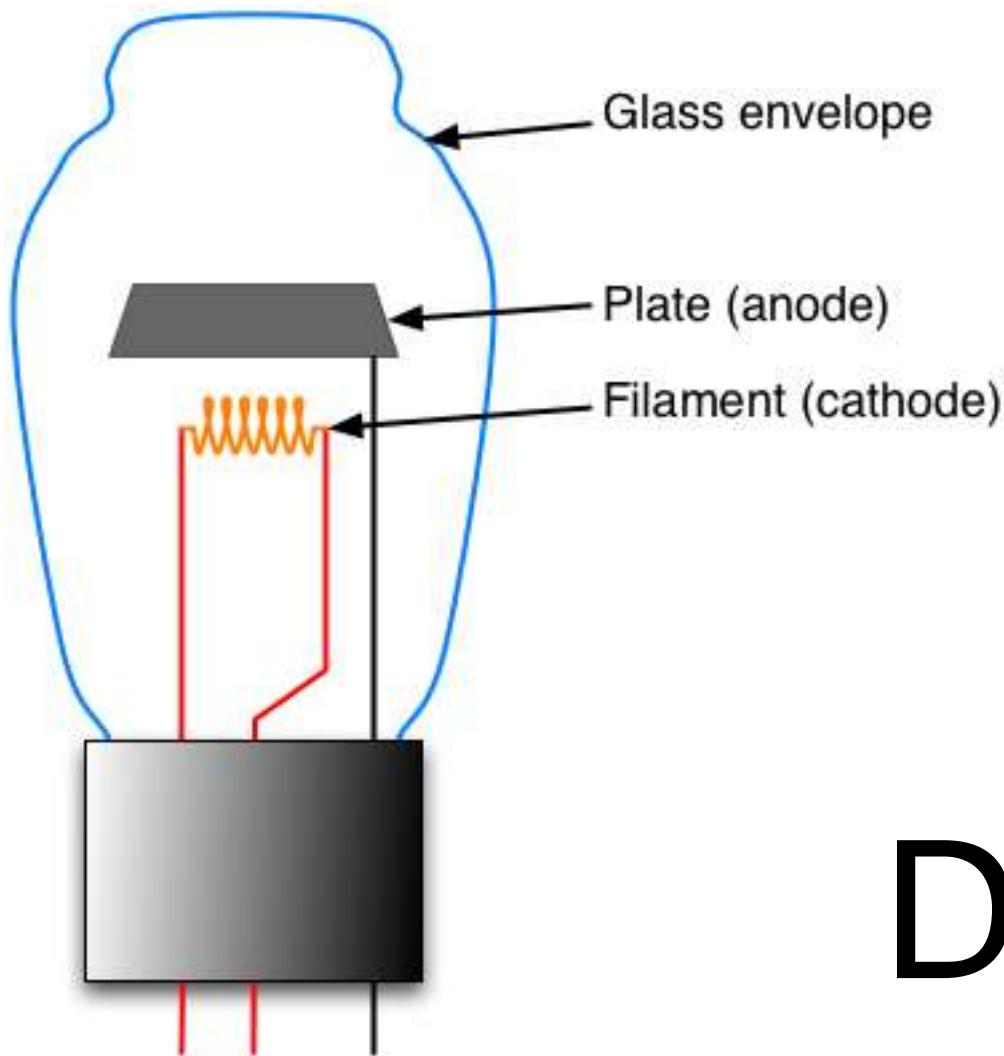
Signature of Chairman of Department,
Committee on Graduate Students _____



But...

- Relays are
 - Slow
 - Good for hundreds of thousands of cycles
 - Wear out
 - Big
 - Noisy

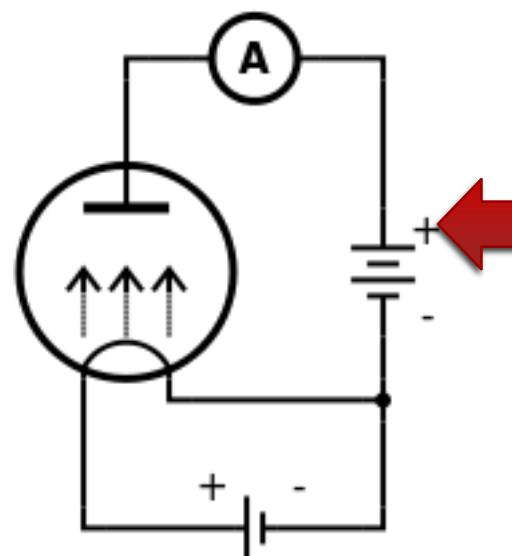
- So...



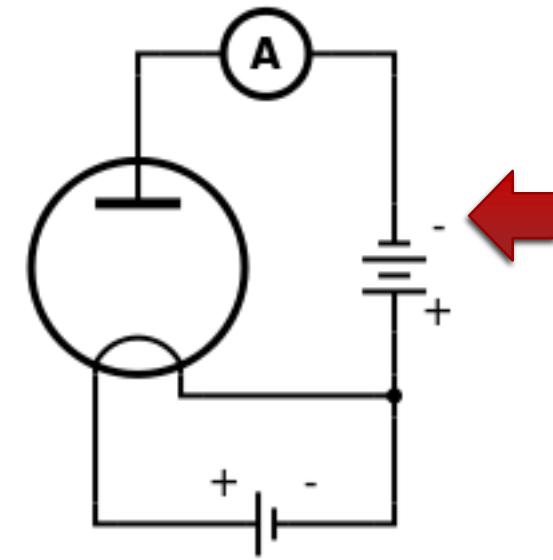
Diode

Edison Effect (Thermionic Emission)

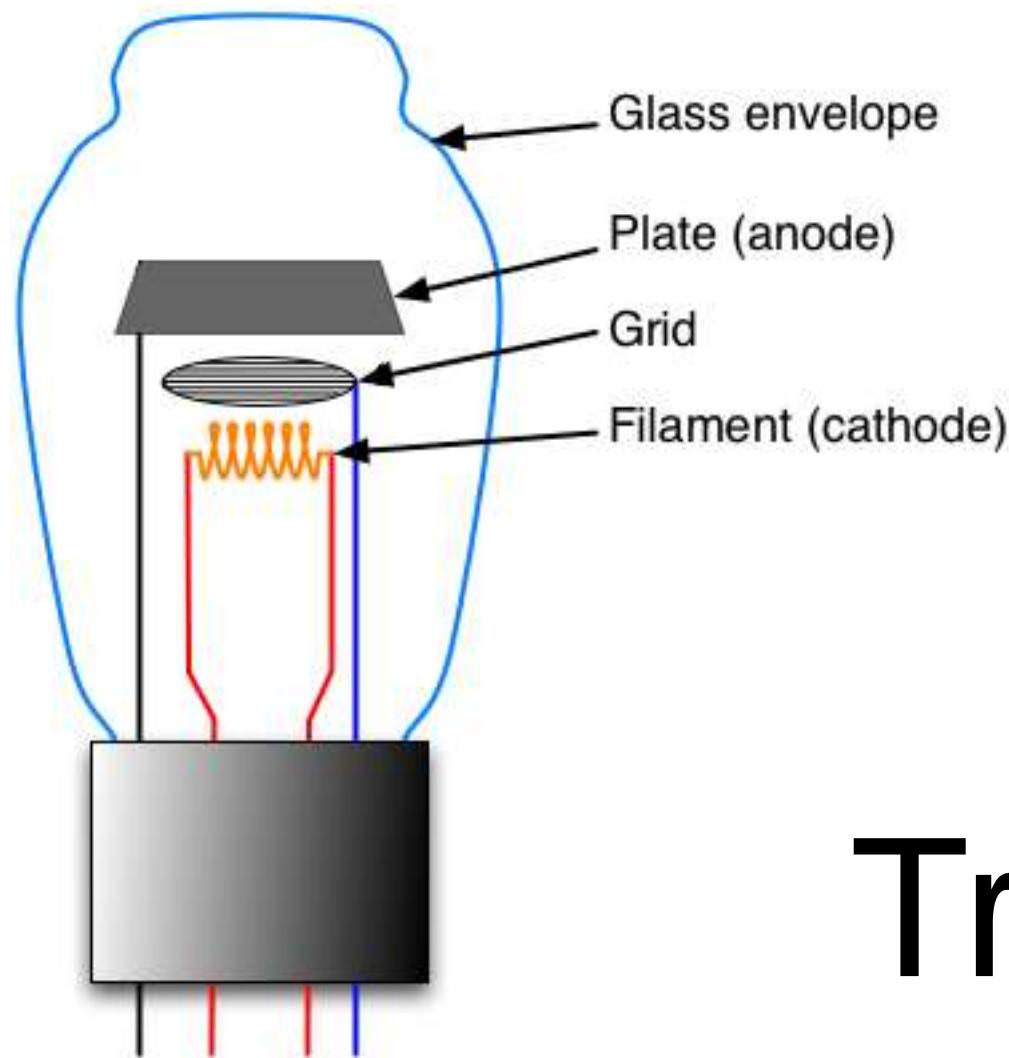
- Note that the electrons will only flow in one direction!
- If an AC power source replaces the anode battery, the current changes polarity twice each cycle
- Current will only flow during the parts of the sine wave where the charge is negative
- The resulting current is thus DC with current flowing every half-cycle
- We call this diode a **rectifier** when it's used to convert AC to DC power



Electron flow



No current

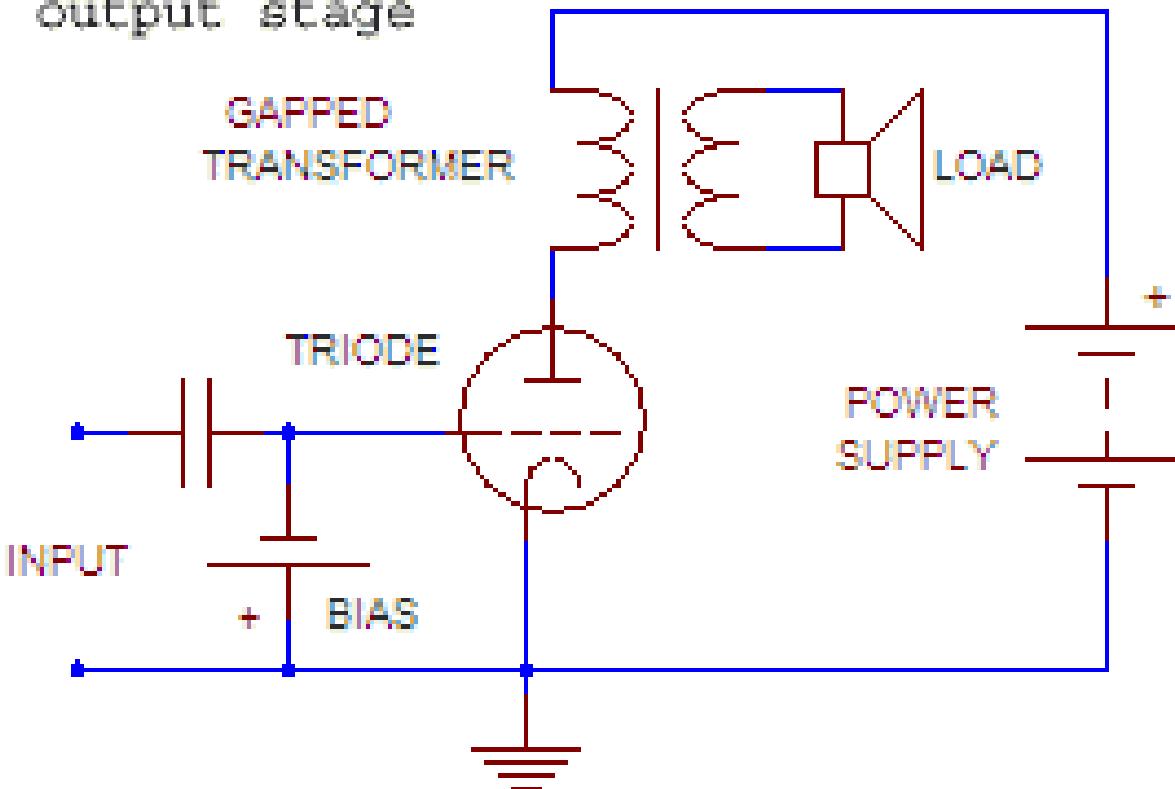


Triode

Triode as Amplifier

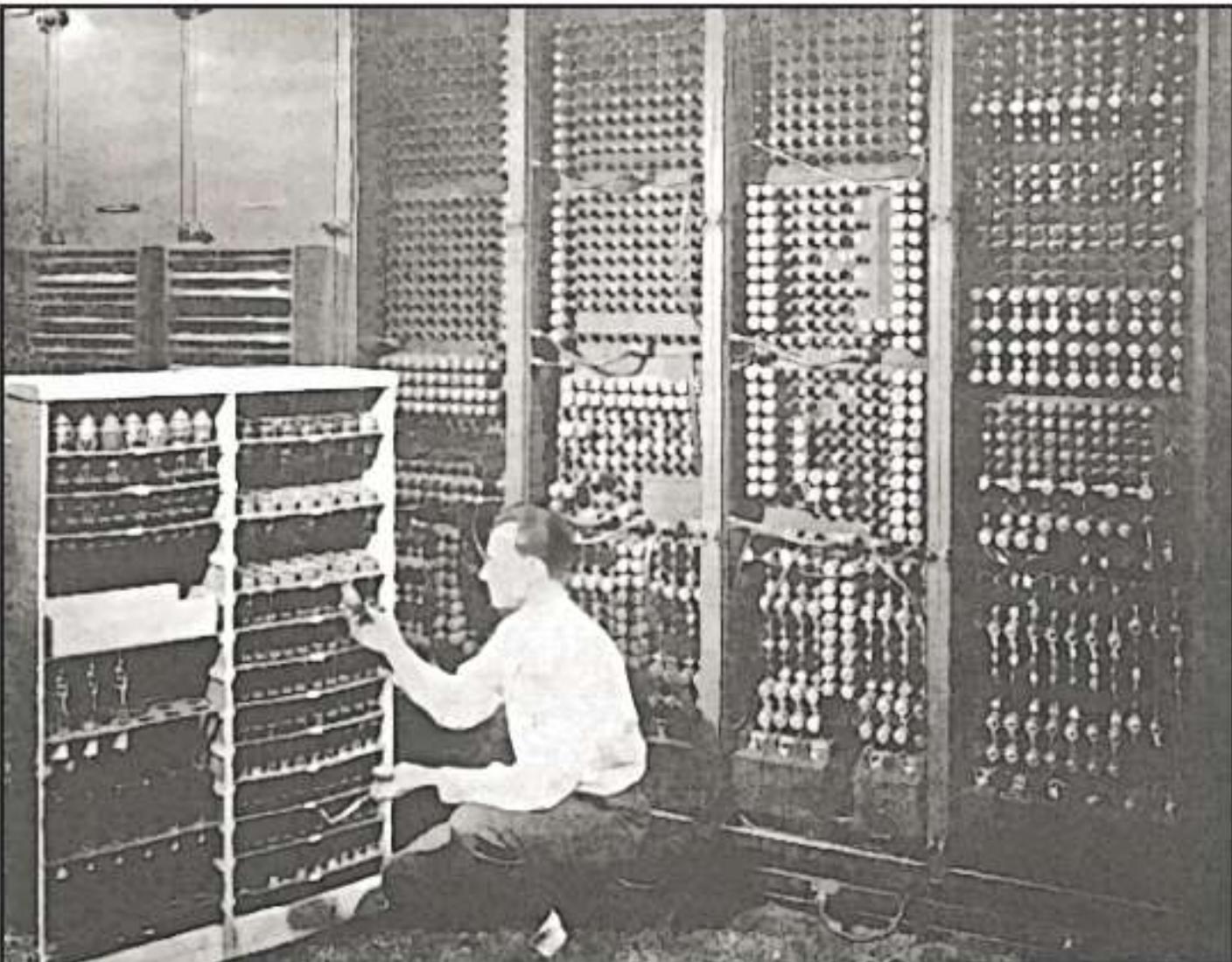
- The input signal is a current from a microphone that represents the sound waves hitting the microphone
- The signal modulates the charge on the triode grid
- The charge applied to the grid permits or inhibits the flow of a larger current from the filament to the anode
- The output signal to the speaker is a higher current but proportional to the input signal

Conventional transformer-coupled output stage



- Used as
 - Switches
 - Amplifiers
- In
 - Radio
 - Television
 - Stereo
 - Radar
 - Computers

eniac vacuum tubes



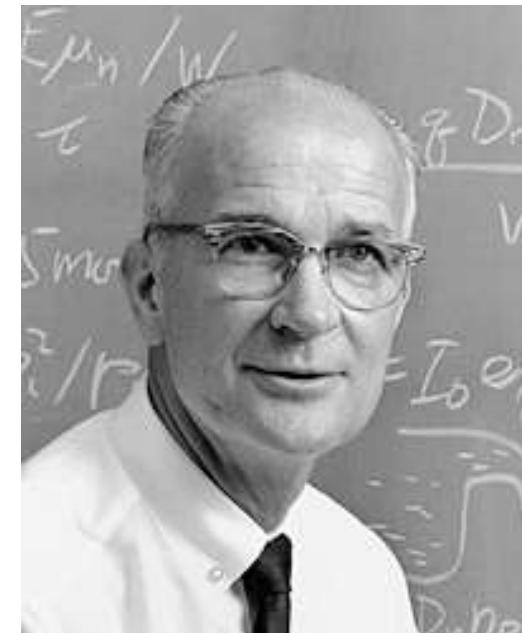
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

But...

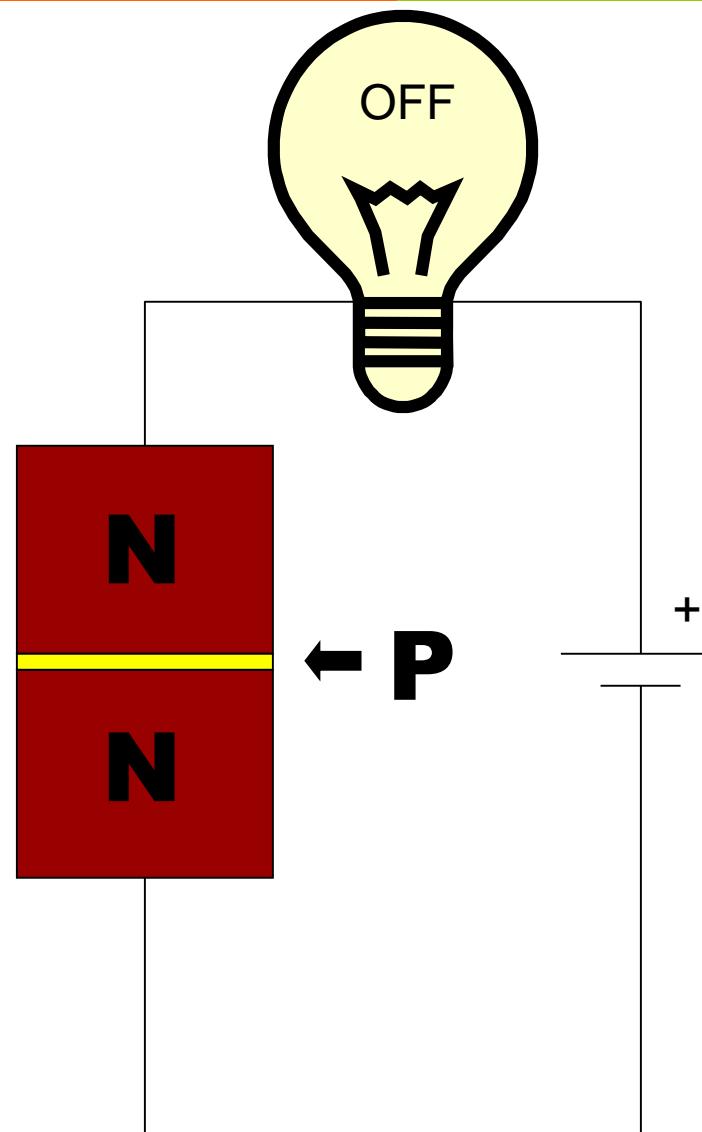
- Tubes are
 - Hot
 - High power consumption
 - High voltages
 - Unreliable
 - Expensive
 - Consist of many individual components
- So...

William Shockley

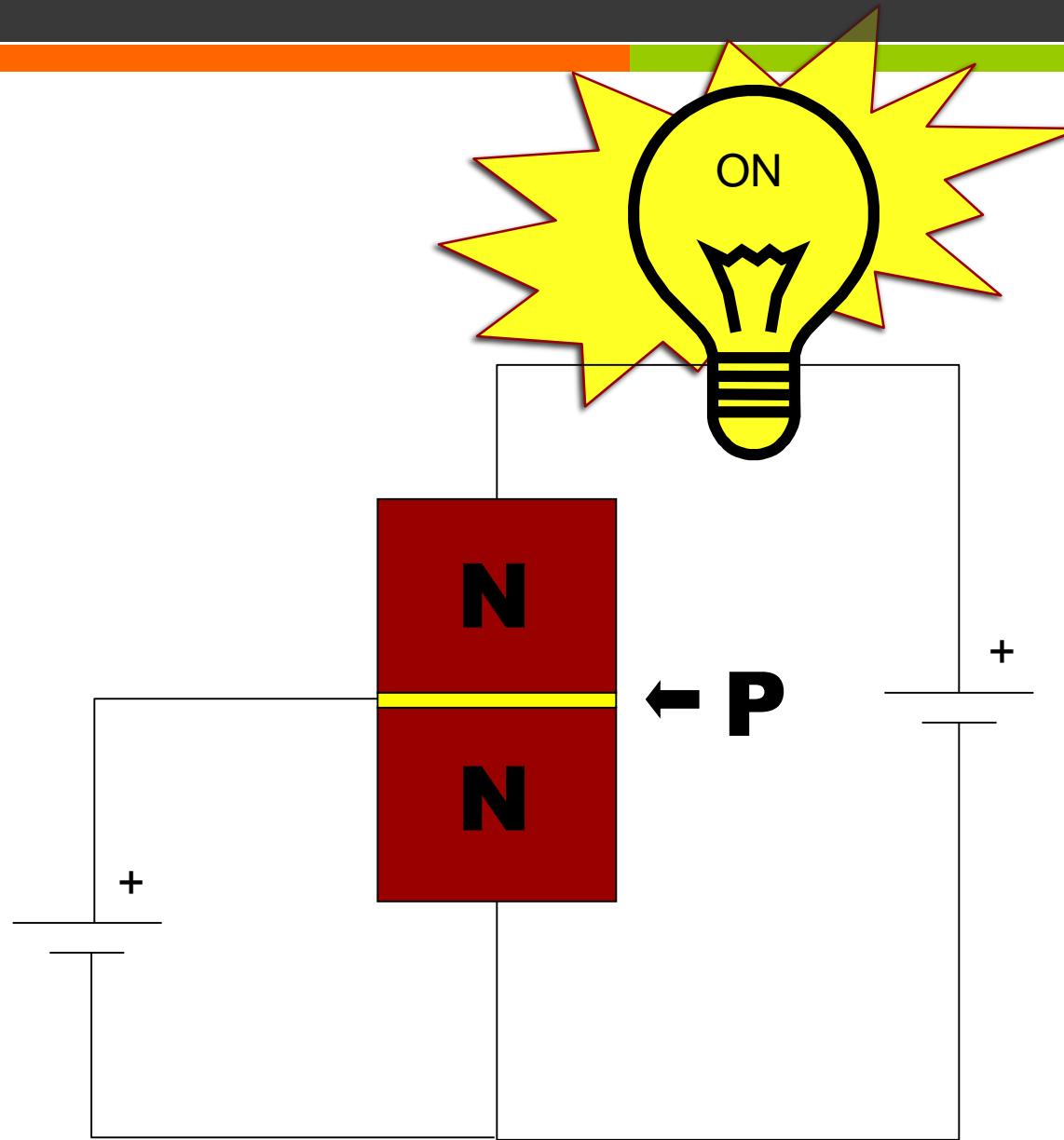
- ↗ Born February 13, 1910 Greater London, England, UK
- ↗ Died August 12, 1989 (aged 79) Stanford, California, US
- ↗ Nationality American
- ↗ Alma mater MIT, Caltech
- ↗ Known for
 - ↗ Point-contact transistor and BJT
 - ↗ Shockley diode equation
 - ↗ Read-Shockley equation
 - ↗ Shockley–Ramo theorem
 - ↗ Haynes–Shockley experiment
 - ↗ Shockley–Queisser limit
- ↗ Awards
 - ↗ Nobel Prize in Physics (1956)
 - ↗ Comstock Prize in Physics (1953)
 - ↗ Oliver E. Buckley Condensed Matter Prize (1953)
 - ↗ Wilhelm Exner Medal (1963)
 - ↗ IEEE Medal of Honor (1980)



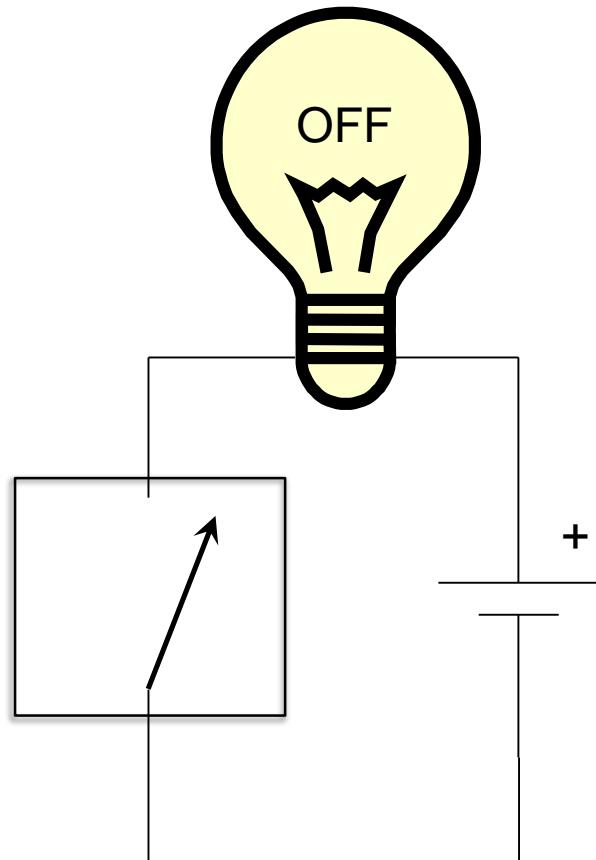
So to save money...



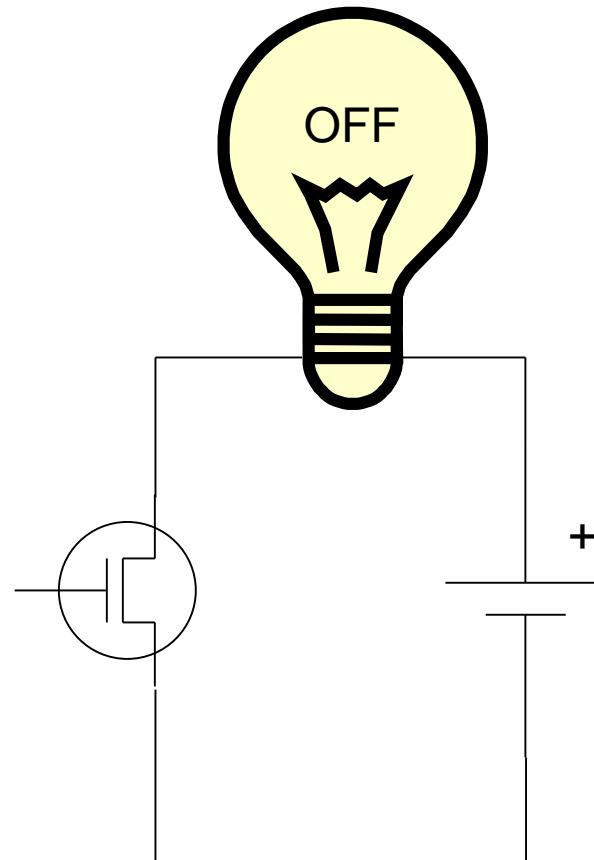
Transistors



Different Switches



A physical switch



An electronic switch

Transistors & Circuits

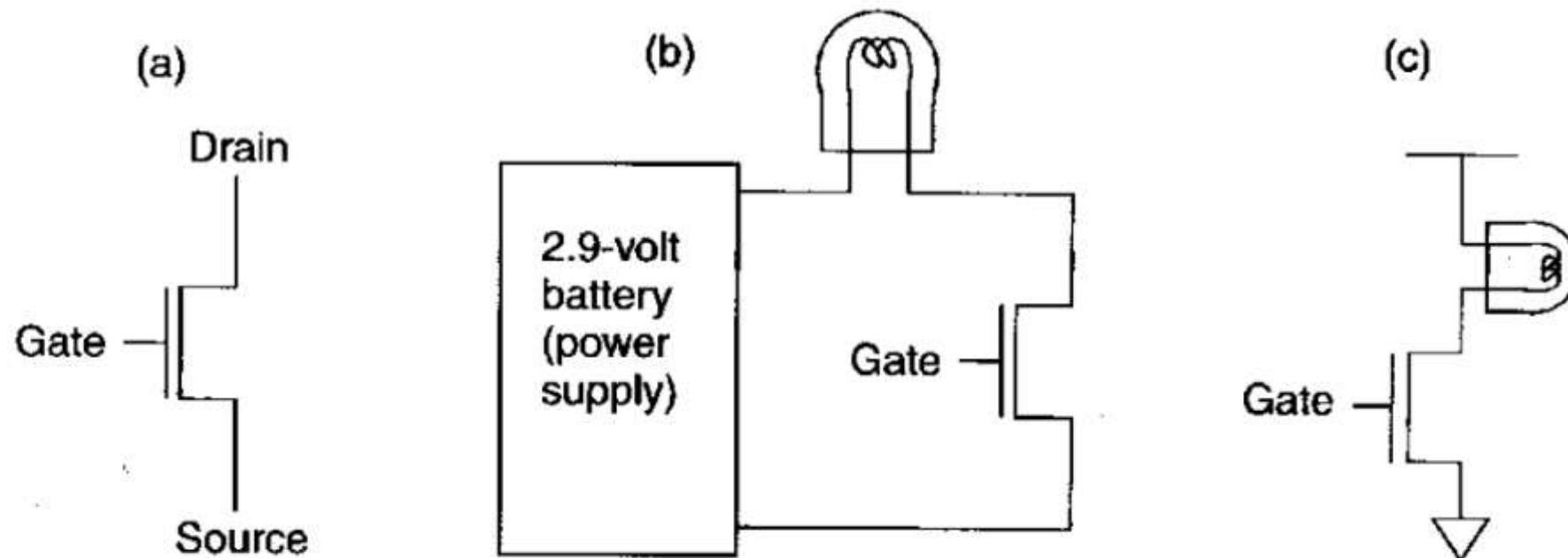
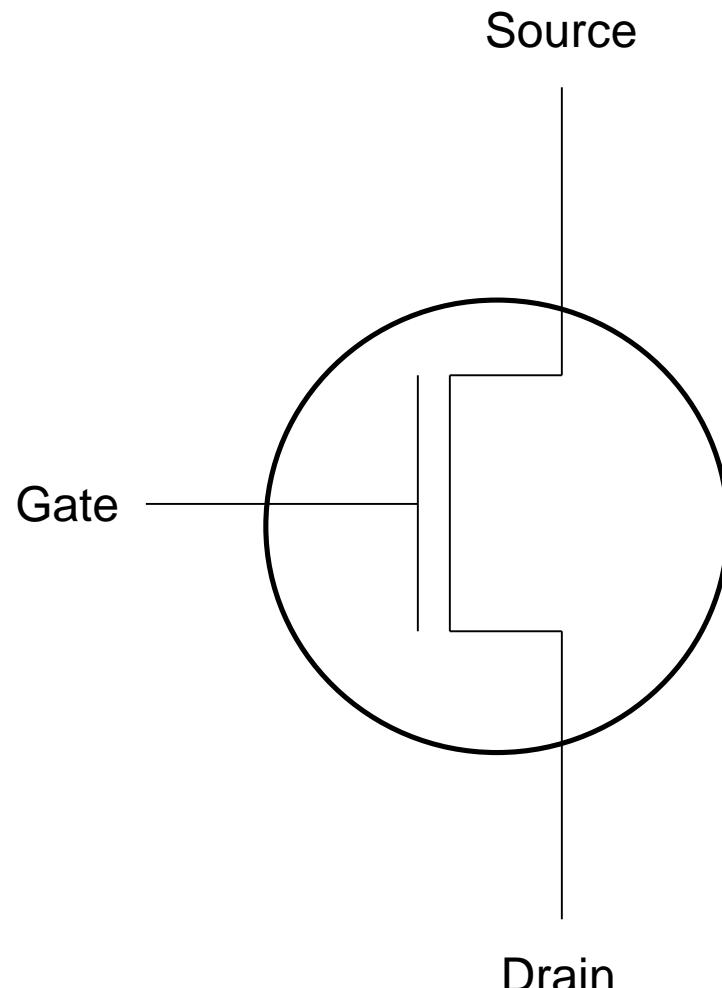
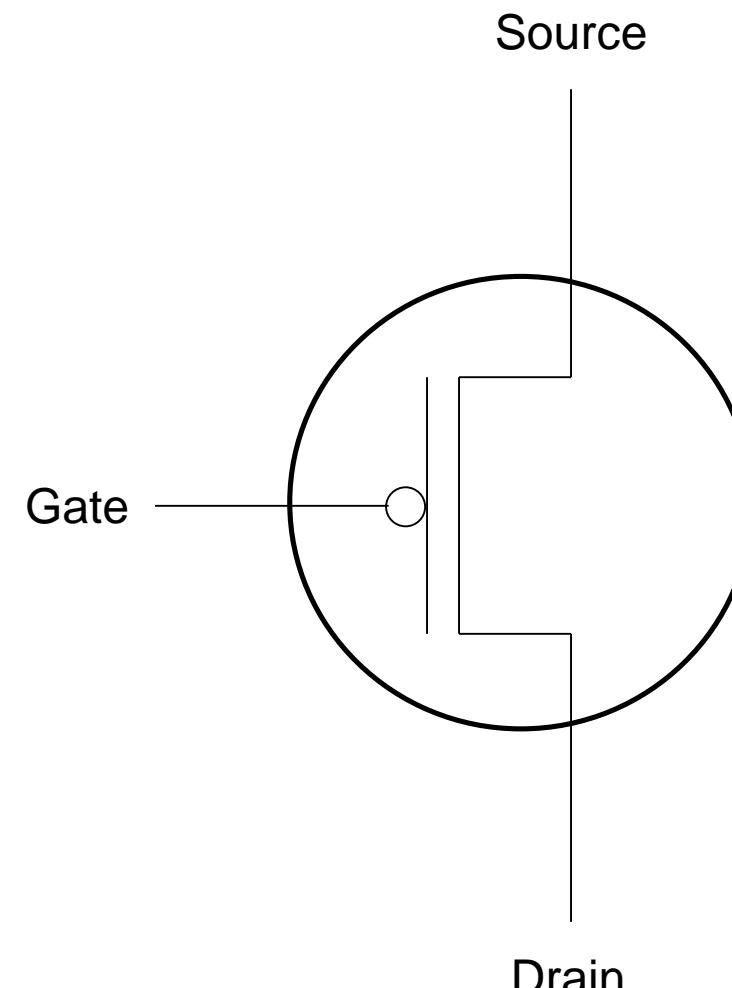


Figure 3.2 The n-type MOS transistor

Complementary Transistors



n-Type MOS



p-Type MOS

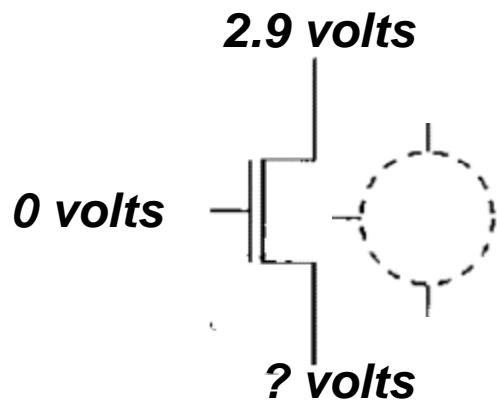
Common Misconception: Digital Wires Have 3 (not 2) States!

- A wire with some designated voltage (e.g. +2.9 volts) can represent a logical 1.
- A wire with some designated voltage (e.g. 0 volts or ground) can represent a logical 0.
- A wire that is not connected to 2.9 volts or ground is said to be **floating** or in a **high impedance state** and its value can randomly vary from a logical 0 to 1.

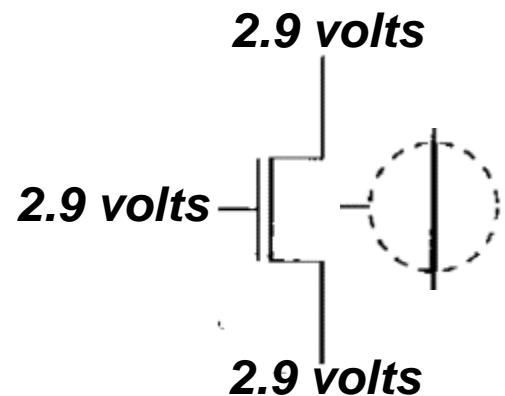
Transistor Comparison

n-Type (normally open)

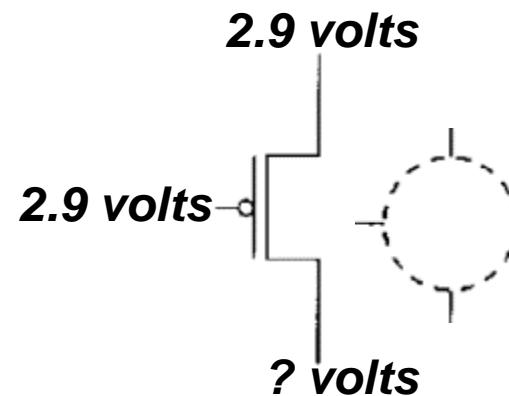
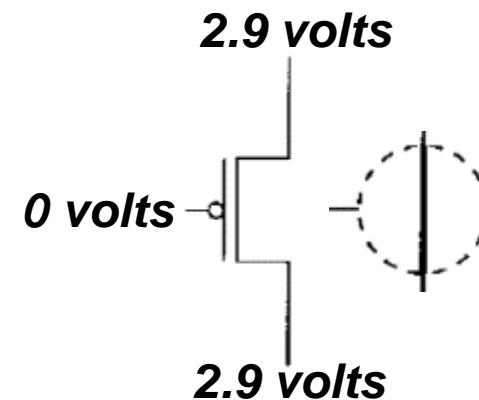
(normally)
without power
to the gate



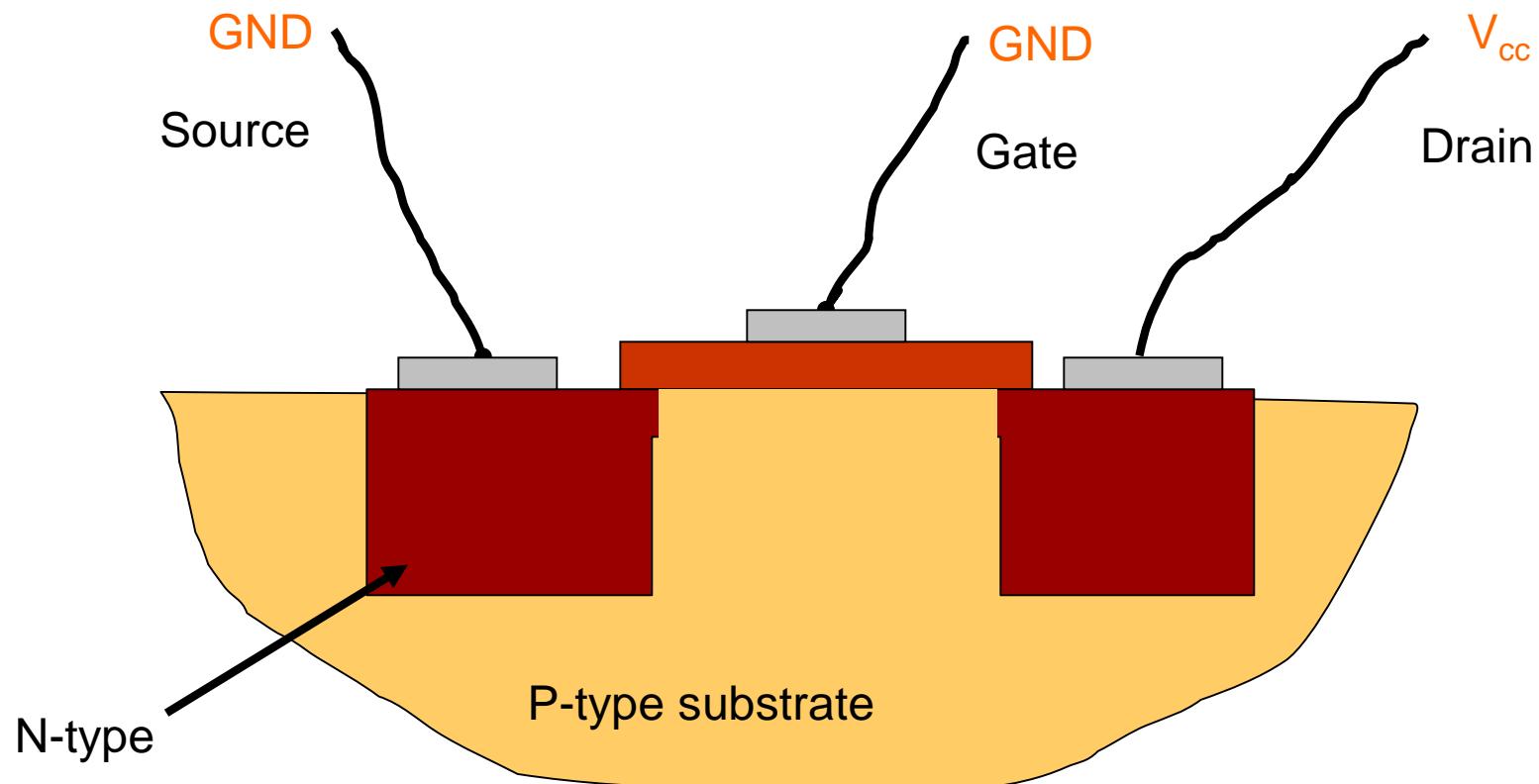
with power
to the gate



p-Type (normally closed)

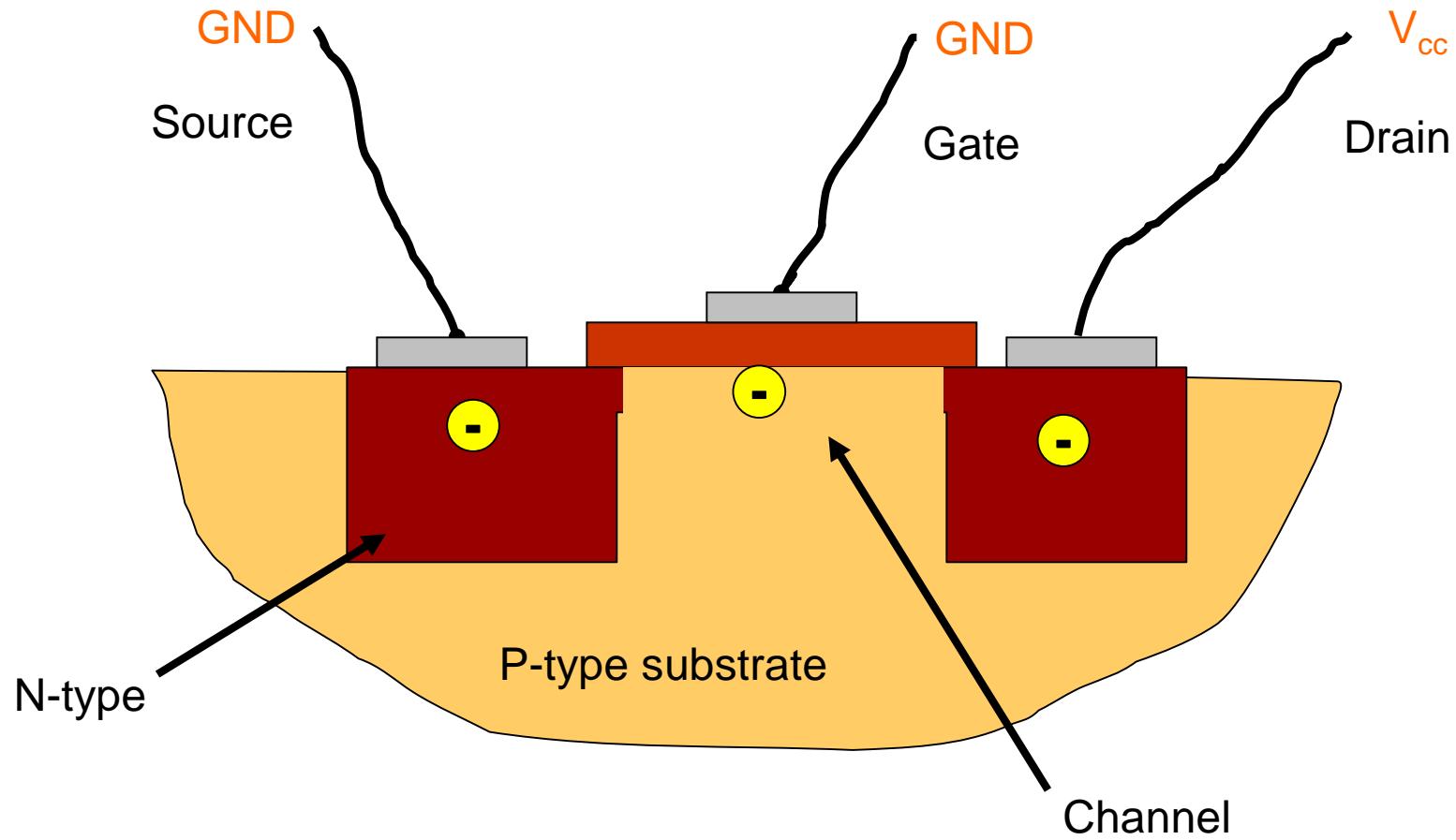


N-type MOS FET*

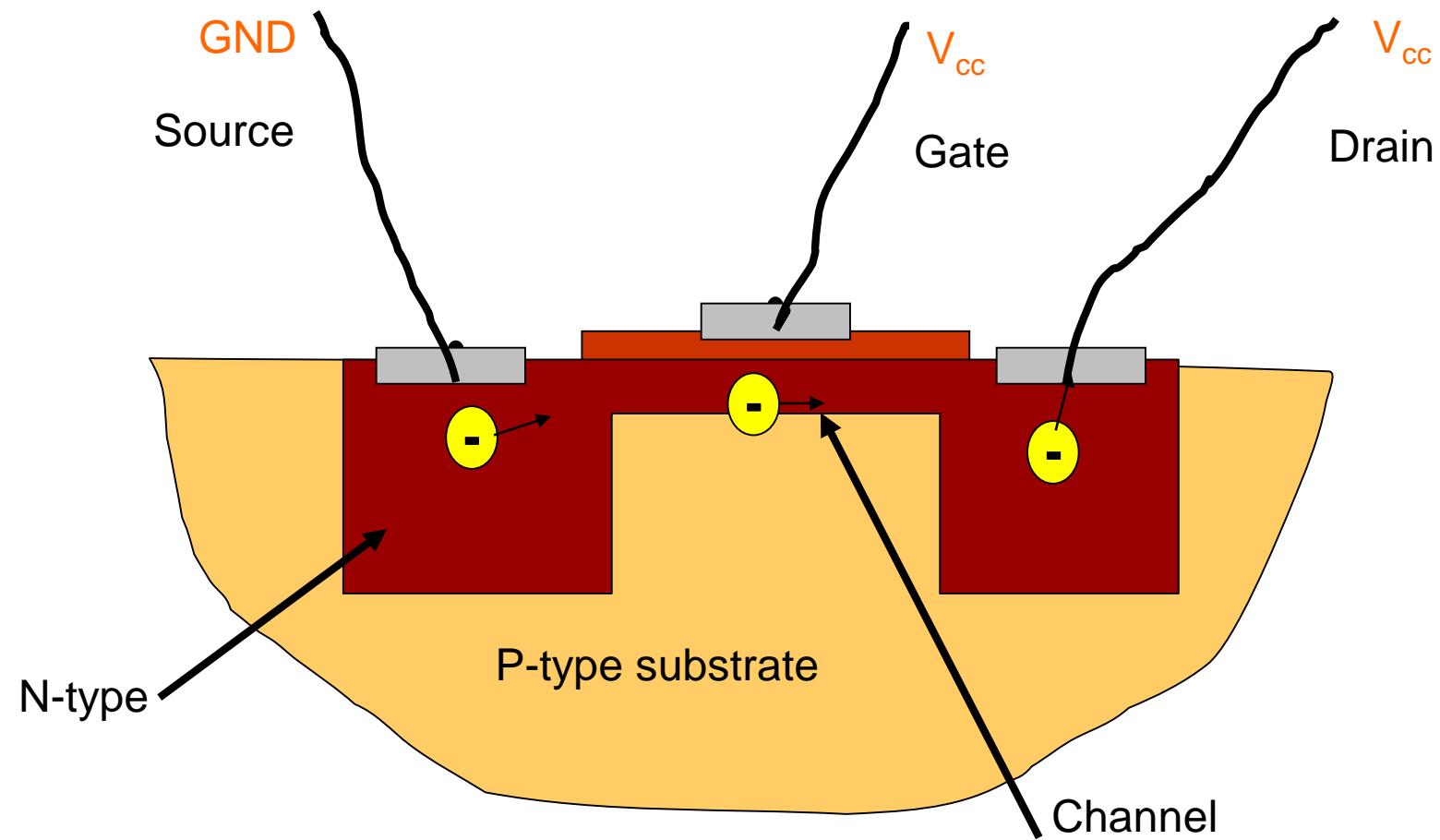


*Metal Oxide Semiconductor Field Effect Transistor

Gate Open

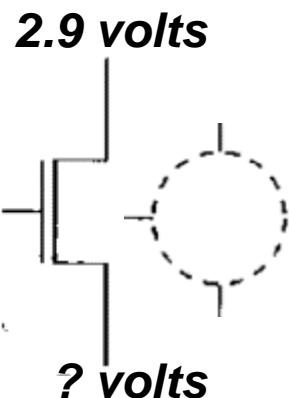


Gate Closed



Question

n-Type (normally open)



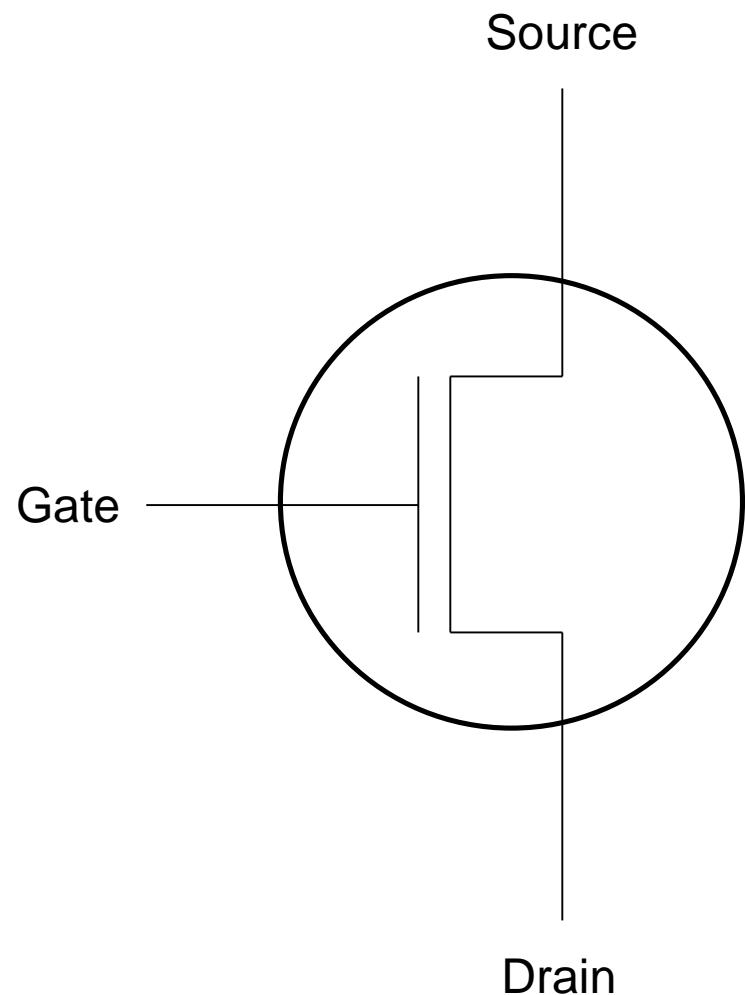
You have an n-type transistor connected to 2.9v and 0v as shown. What voltage value will we measure at the drain?

- A. 0v
- B. 2.9v
- C. 1.45v
- D. Unknown

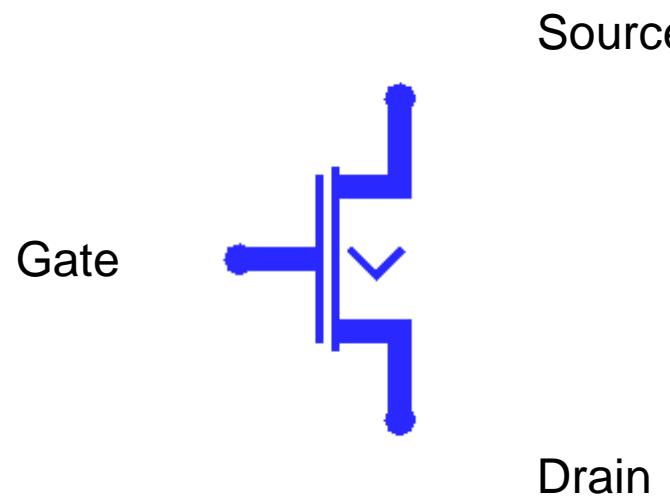
Because the gate is at 0v, the drain is disconnected from the source and is floating. We can't tell what value will be measured.



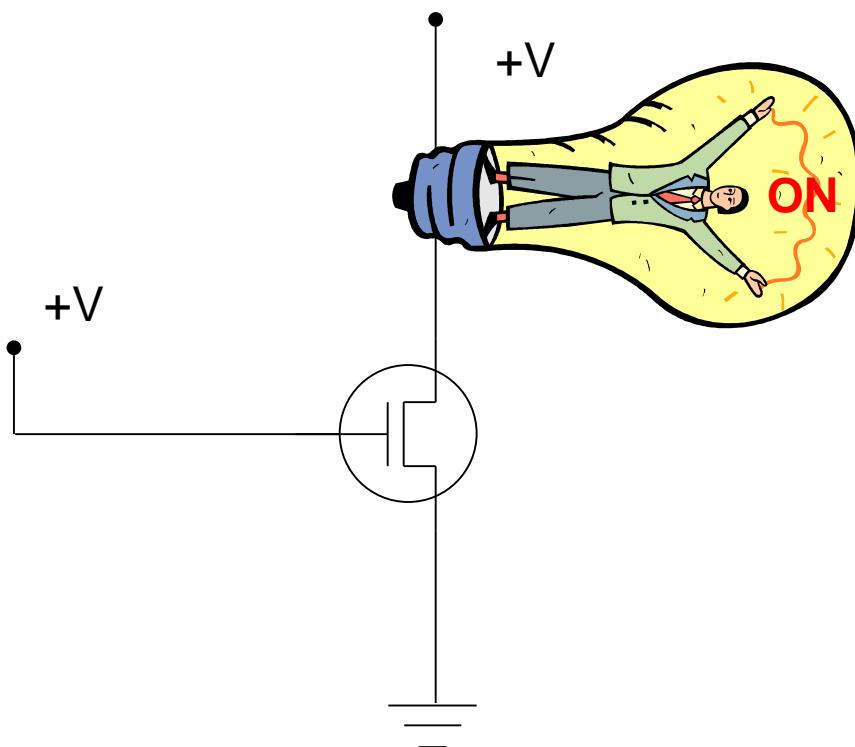
N-Type FET



CircuitSim N-Type MOS FET

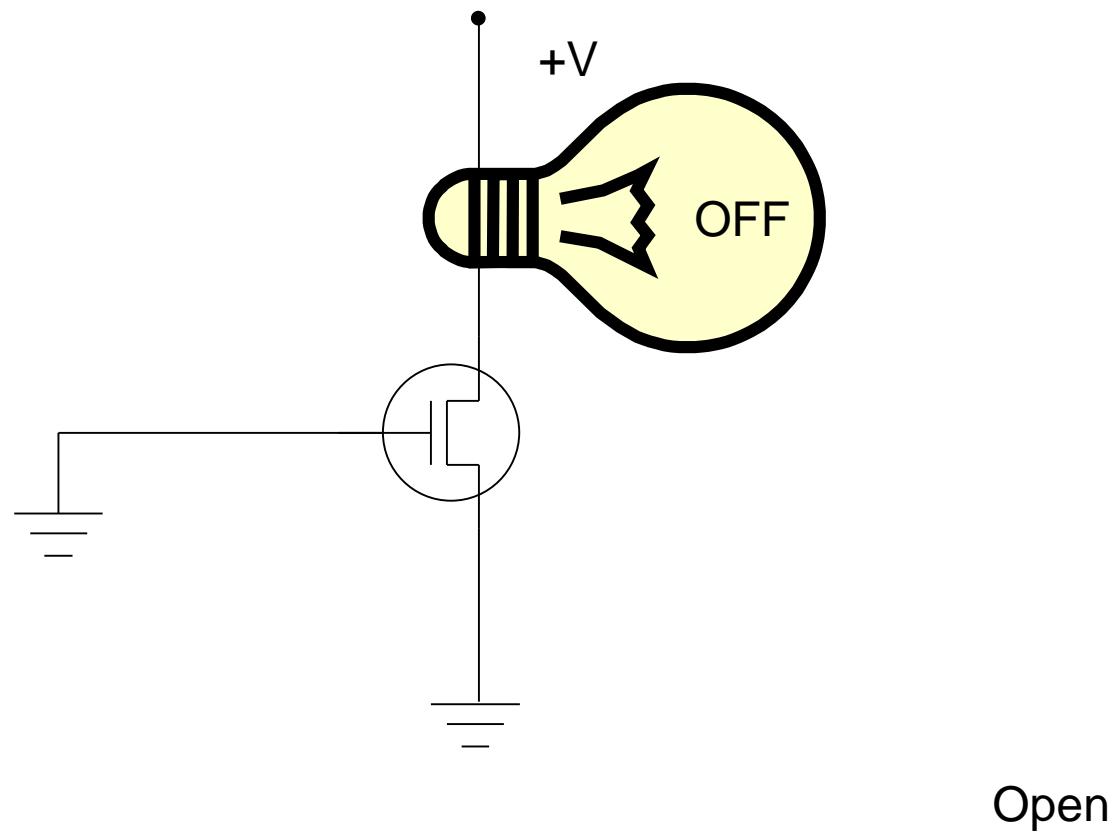


N-Type with Gate at +V

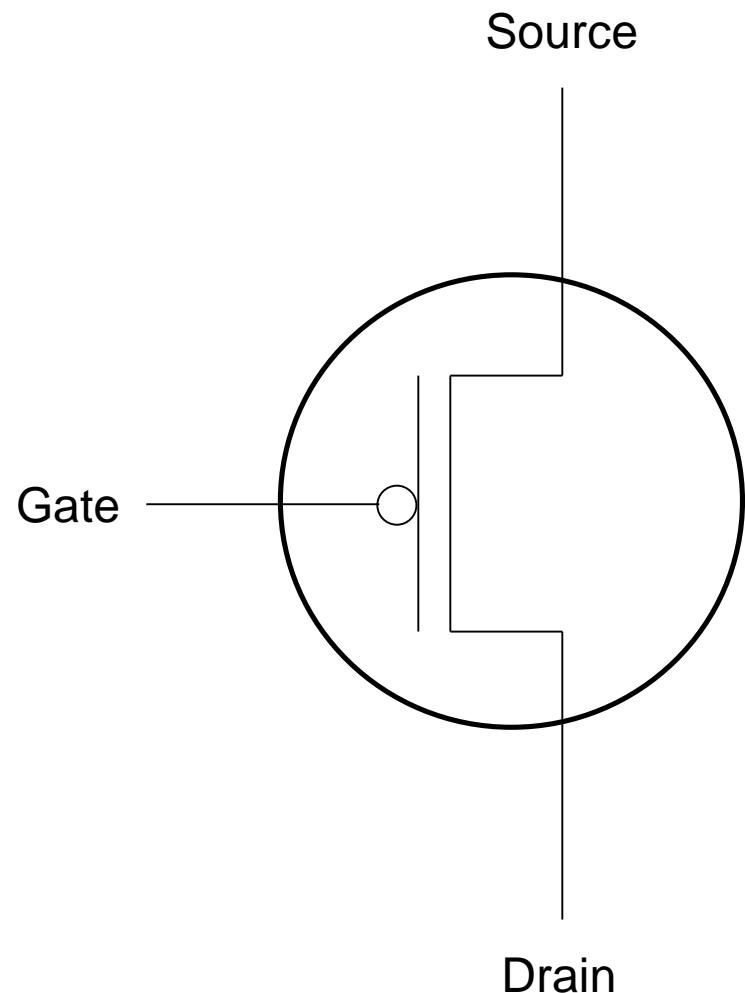


With the supply voltage applied to the device the transistor acts like a closed or connected switch

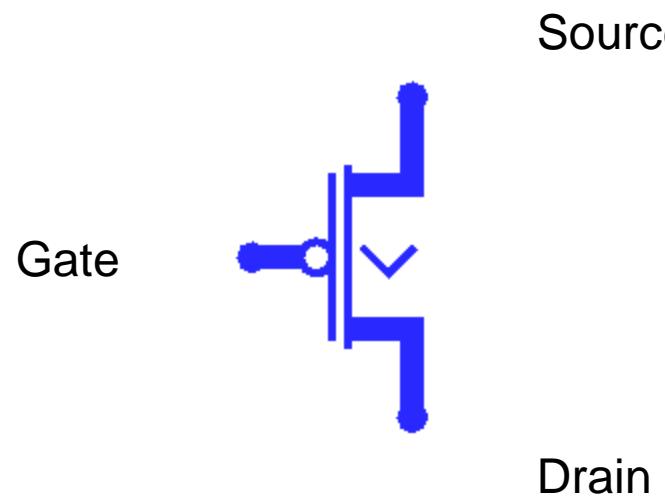
N-Type with Gate at Ground



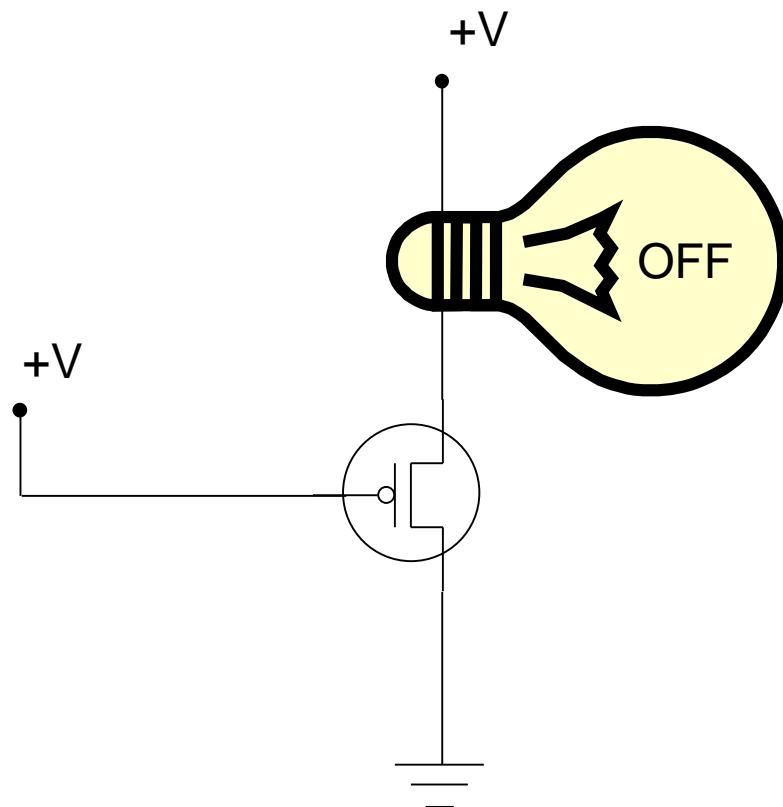
P-Type MOS FET



CircuitSim P-Type MOS FET

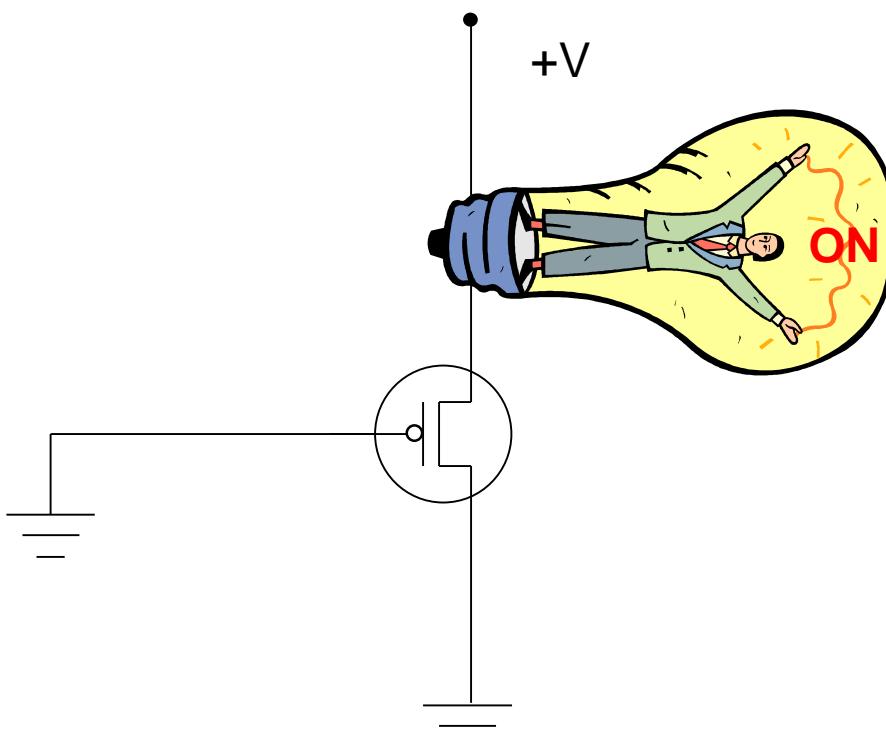


P-Type with Gate at +V



With the supply voltage applied to the device the transistor acts like a open or disconnected switch

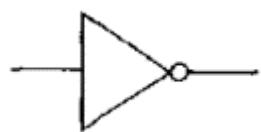
P-Type with Gate at Ground



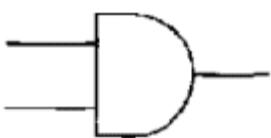
Closed

Logical Operations Revisited

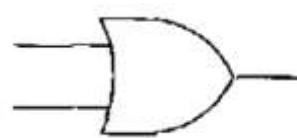
A	NOT
0	1
1	0



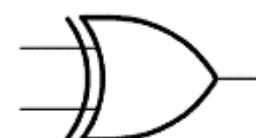
A	B	AND
0	0	0
0	1	0
1	0	0
1	1	1



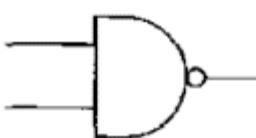
A	B	OR
0	0	0
0	1	1
1	0	1
1	1	1



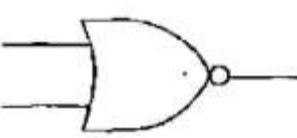
A	B	XOR
0	0	0
0	1	1
1	0	1
1	1	0



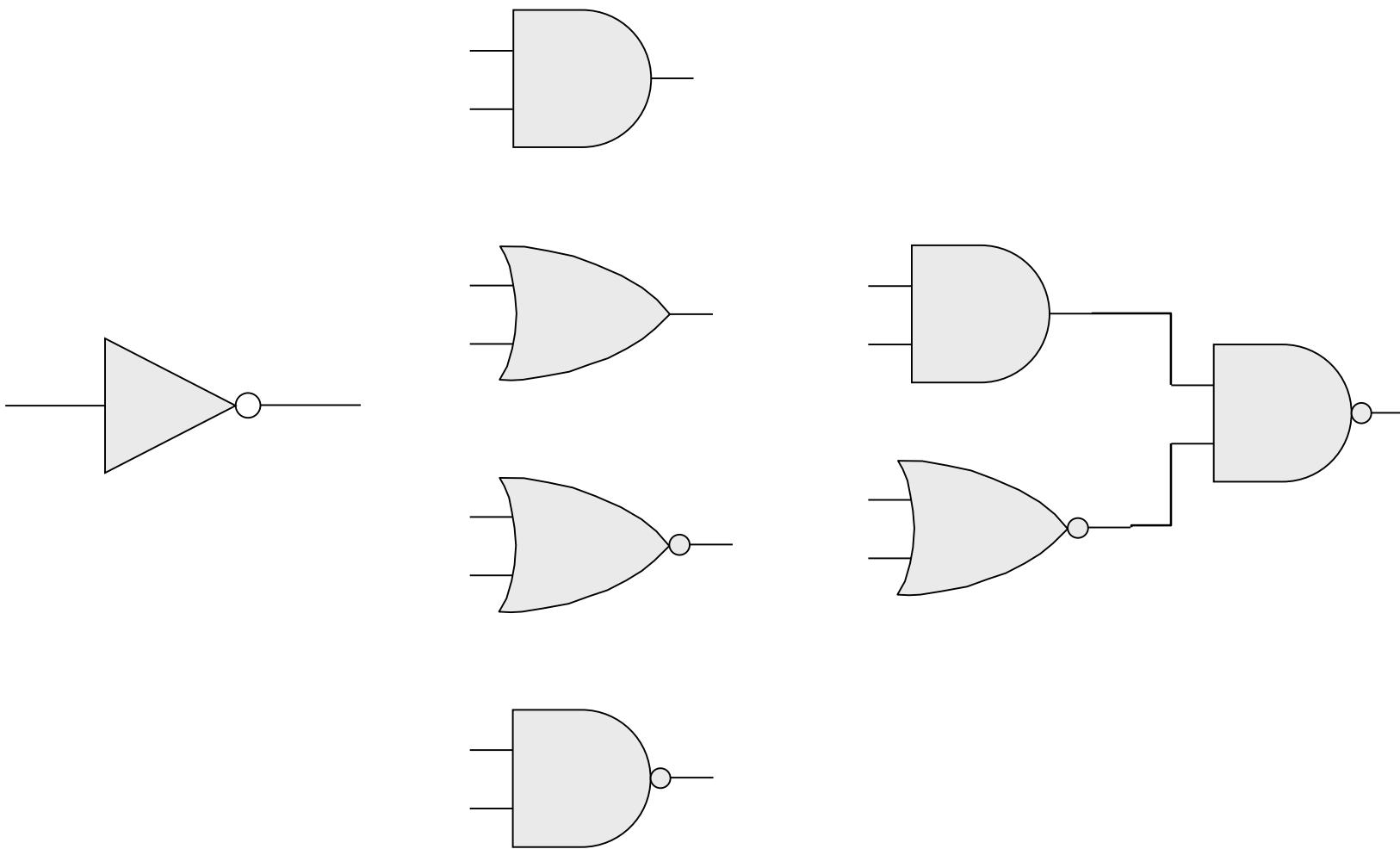
A	B	NAND
0	0	1
0	1	1
1	0	1
1	1	0



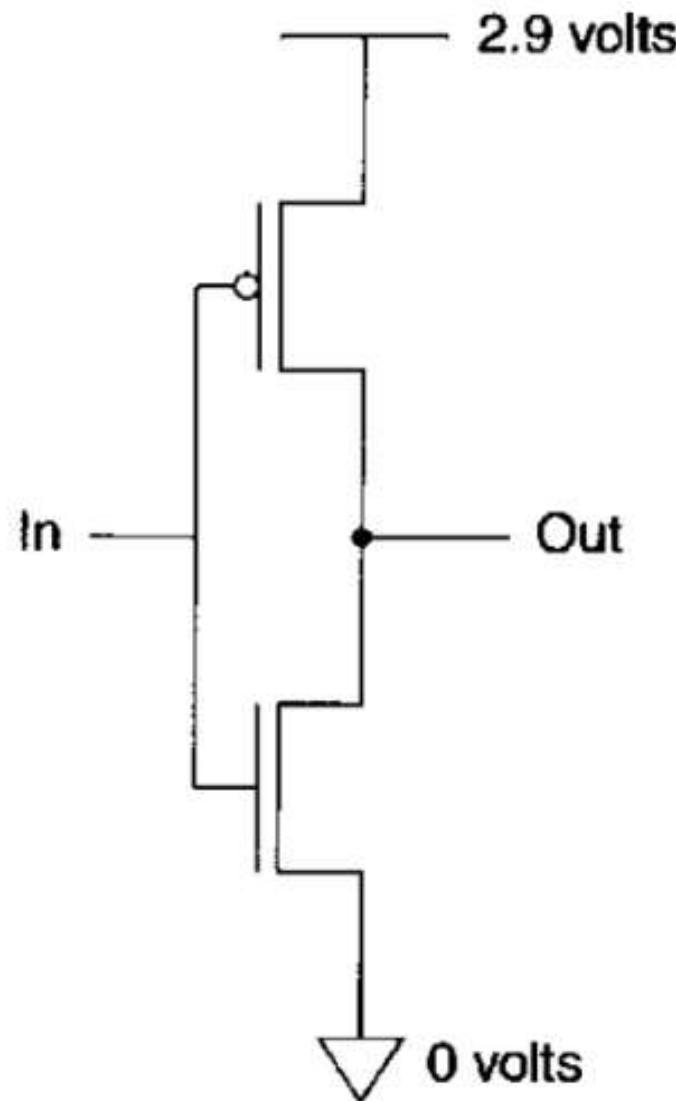
A	B	NOR
0	0	1
0	1	0
1	0	0
1	1	0



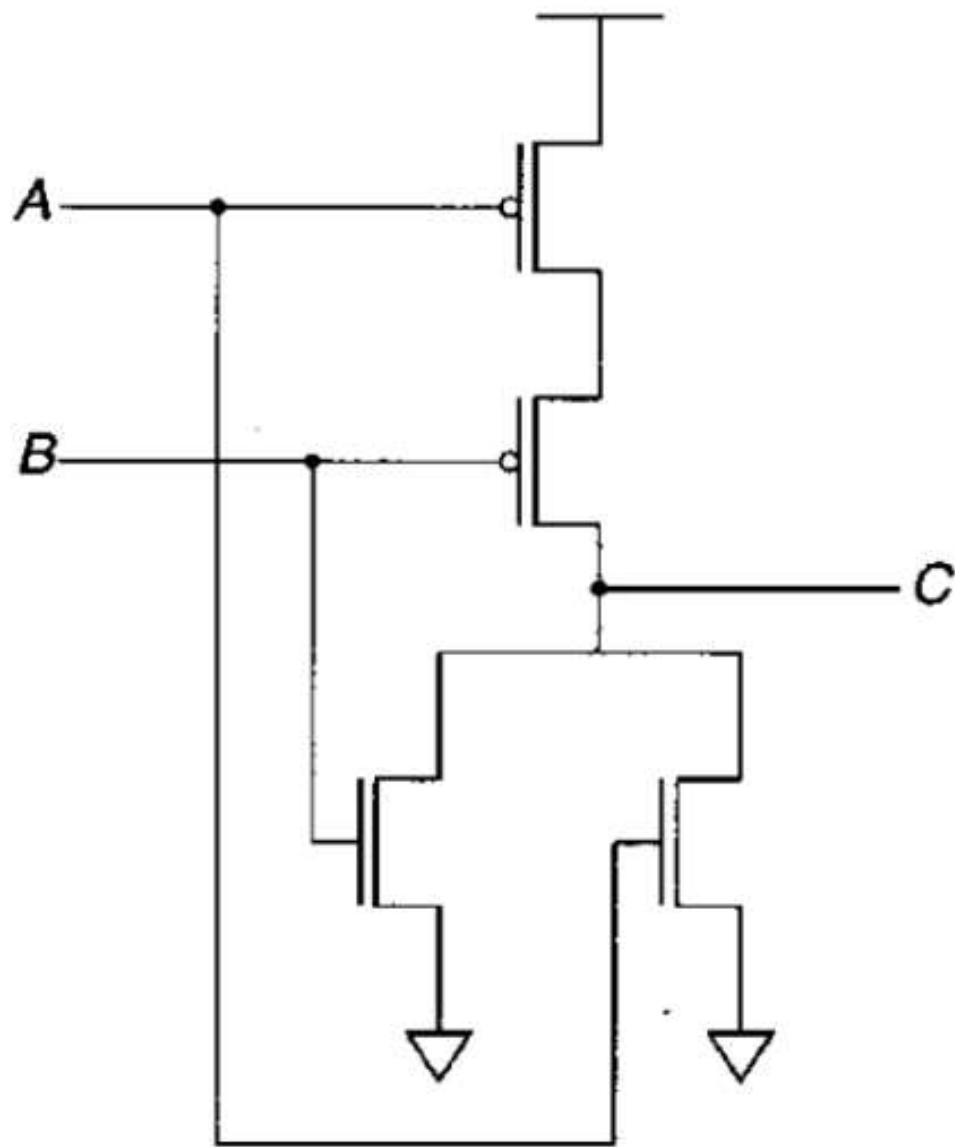
Boolean Gate Representation



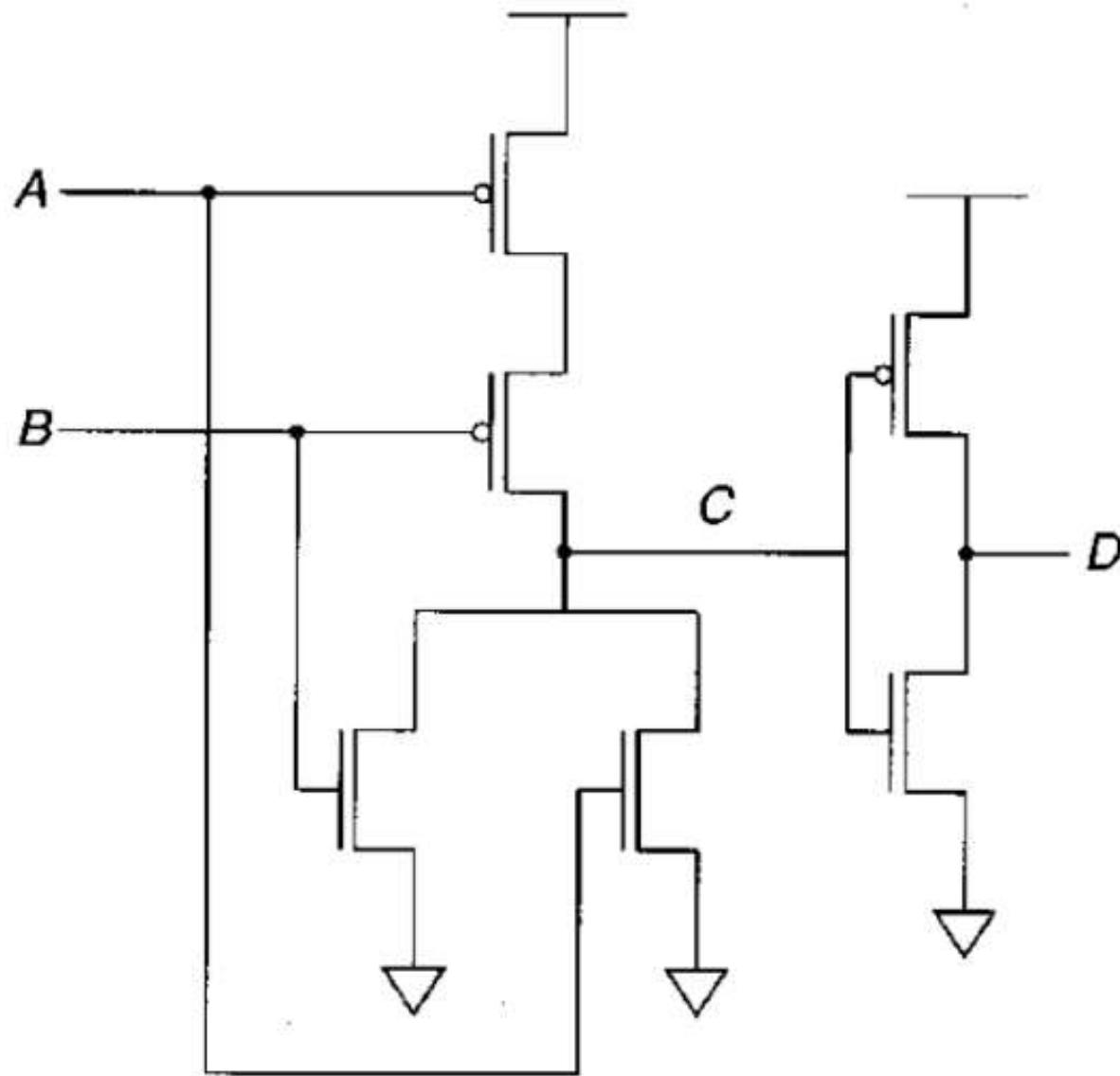
NOT Gate (Inverter)



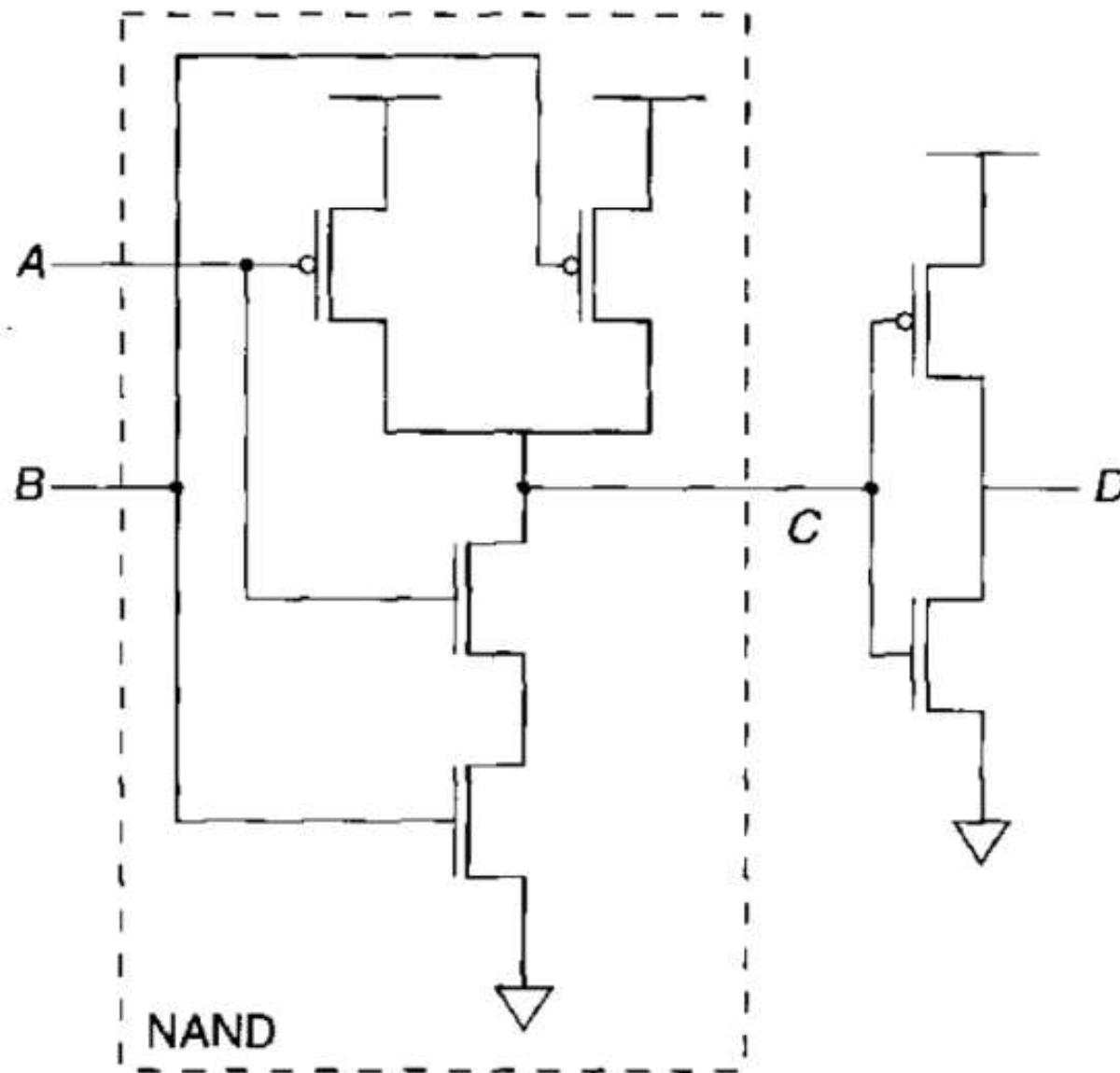
NOR Gates...



...OR Gates



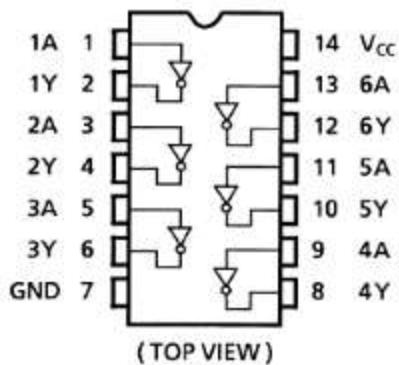
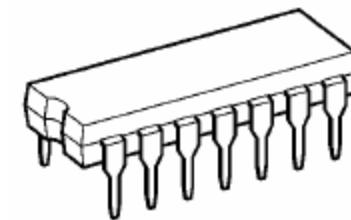
NAND and AND Gates



- ↗ Discrete transistors
 - ↗ Are still big
 - ↗ Use a lot of material to build
 - ↗ Use a lot of power (and hence generate a lot of heat); however smaller transistors need smaller amounts of power
 - ↗ Need a lot of individual connections, all of which are prone to failure
 - ↗ Are so useful that we need *lots* of them
- ↗ So...

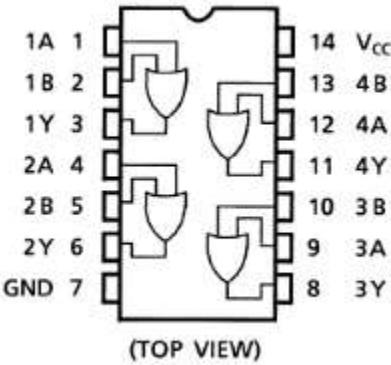
Digital Logic: Building Blocks

- The basic digital logic building blocks are available as pre-packaged self-contained integrated circuits.
- These chips are fast, inexpensive and easy to work with.
- Each chip contains 1-6 gates. Complicated designs can require thousands of gates and hundreds of chips.



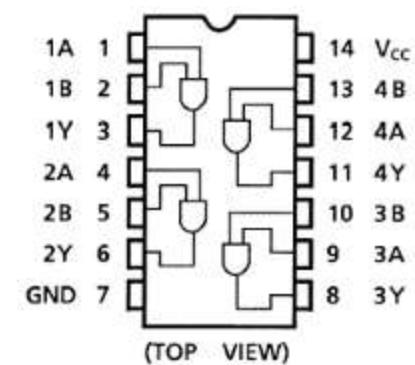
6 NOT Gates

74HC04



4 OR Gates

74HC32



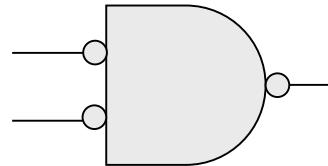
4 AND Gates

74HC08

DeMorgan's Law

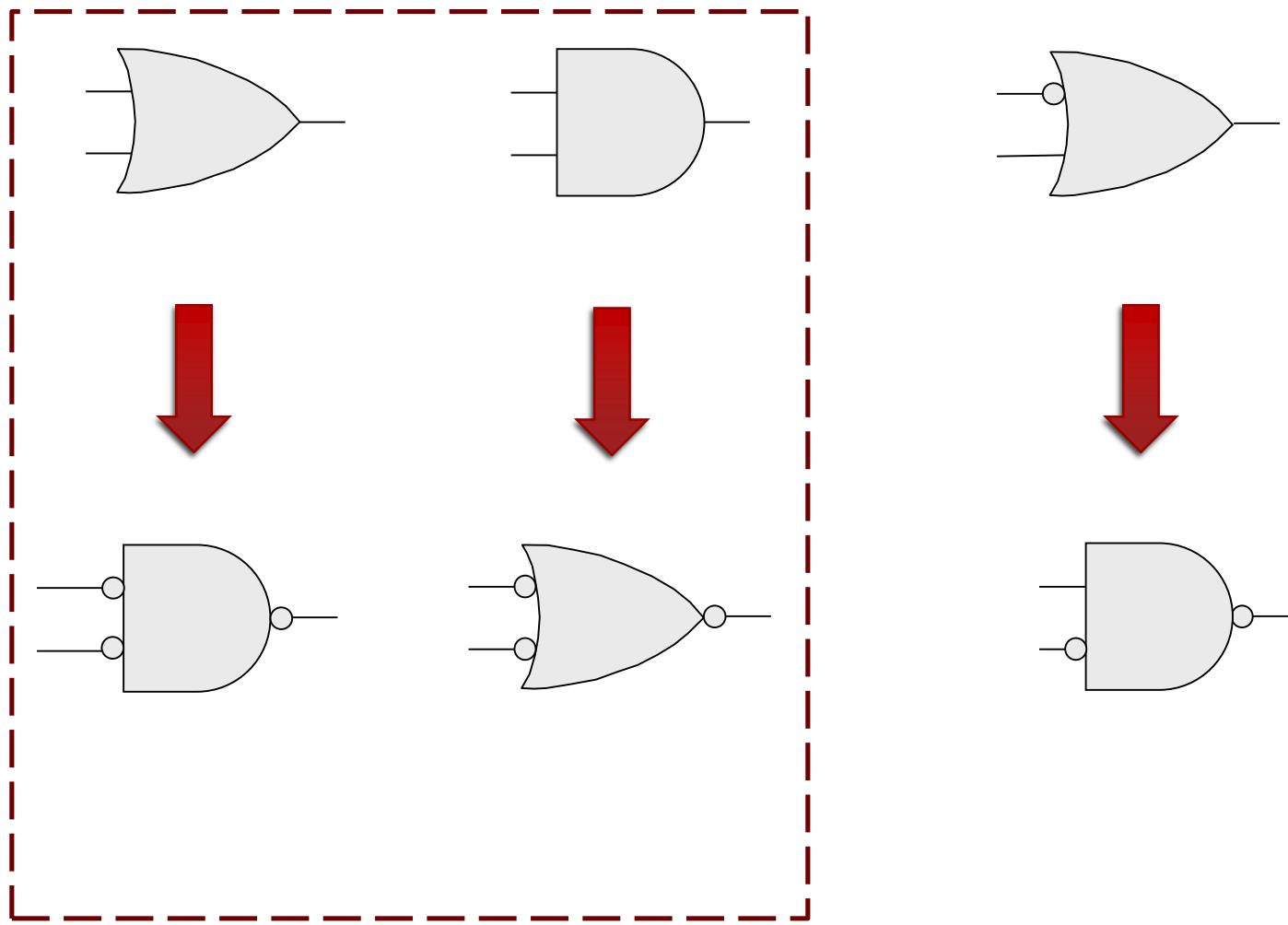
$$\Rightarrow (A'B')' = A+B$$

$$\Rightarrow (A'+B')' = AB$$

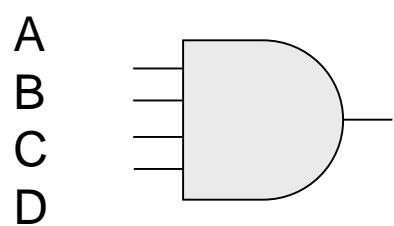


A	B	A'	B'	A'B'	(A'B')'	A'+B'	(A'+B')'	A+B	AB
0	0	1	1	1	0	1	0	0	0
0	1	1	0	0	1	1	0	1	0
1	0	0	1	0	1	1	0	1	0
1	1	0	0	0	1	0	1	1	1

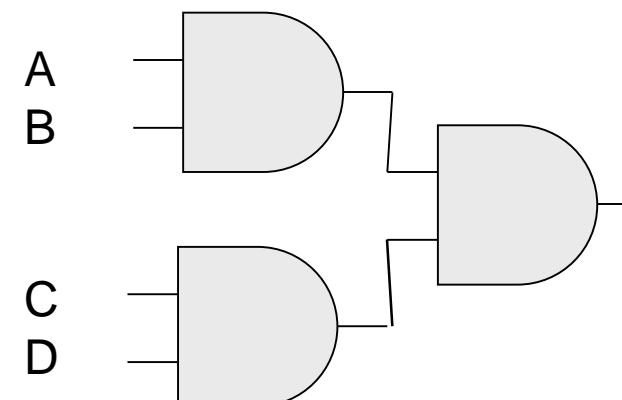
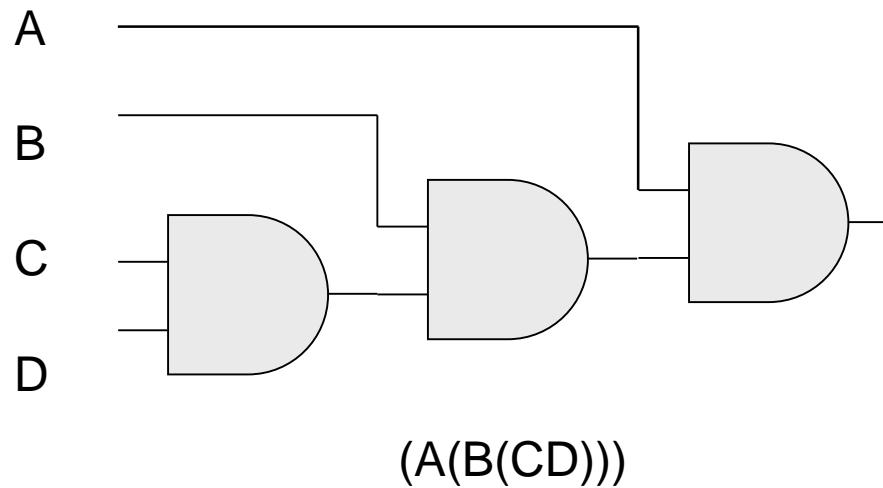
Conte Bubble Theorem



Larger Gates

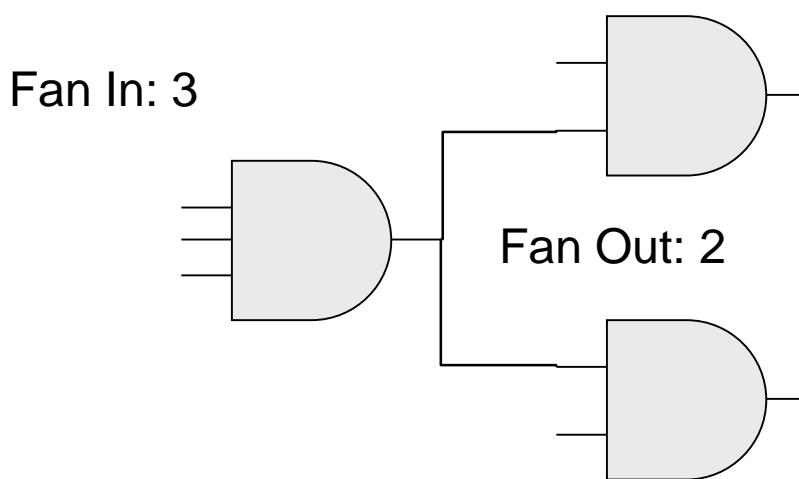


ABCD



$((AB)(CD))$

Fan-In and Fan-Out



Warning

↗ Important concept approaching!

Combinational Logic

- A combination of AND, OR, NOT (plus NAND & NOR)
- The same inputs always produce same output

Whack-A-Mole!



Fun!



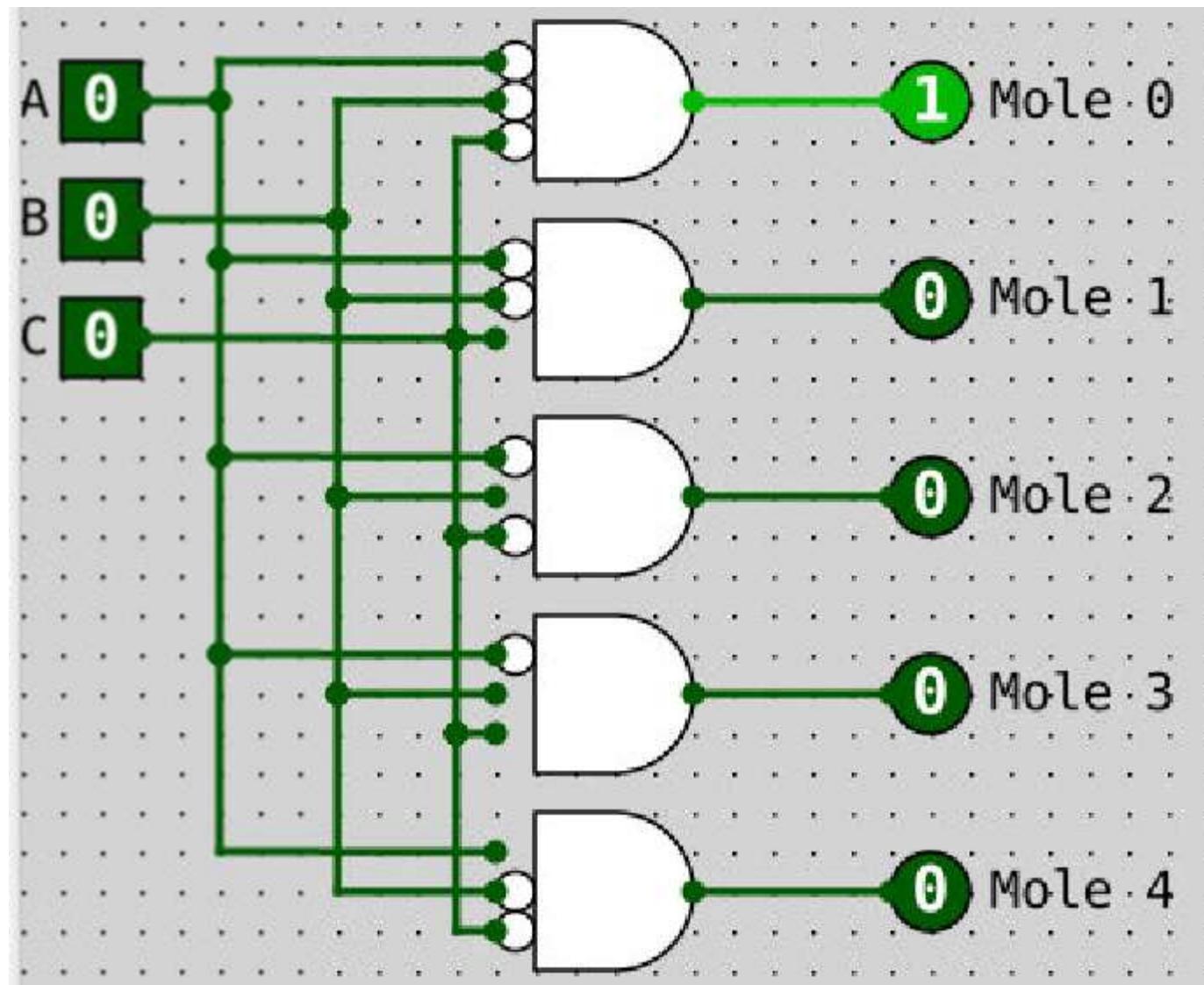


Problem

- You have an n bit binary number which signifies which mole is selected. How do we electrically actuate the desired mole?



Decoder



Question

If you had a decoder with a 5 bit input (i.e. a 5 bit binary number) how many outputs at most could the decoder have?

2^5 possible
input values

So a maximum
of $2^5 = 32$
output values

- A. 5
- B. 10
- C. 16
- D. 32 
- E. 64

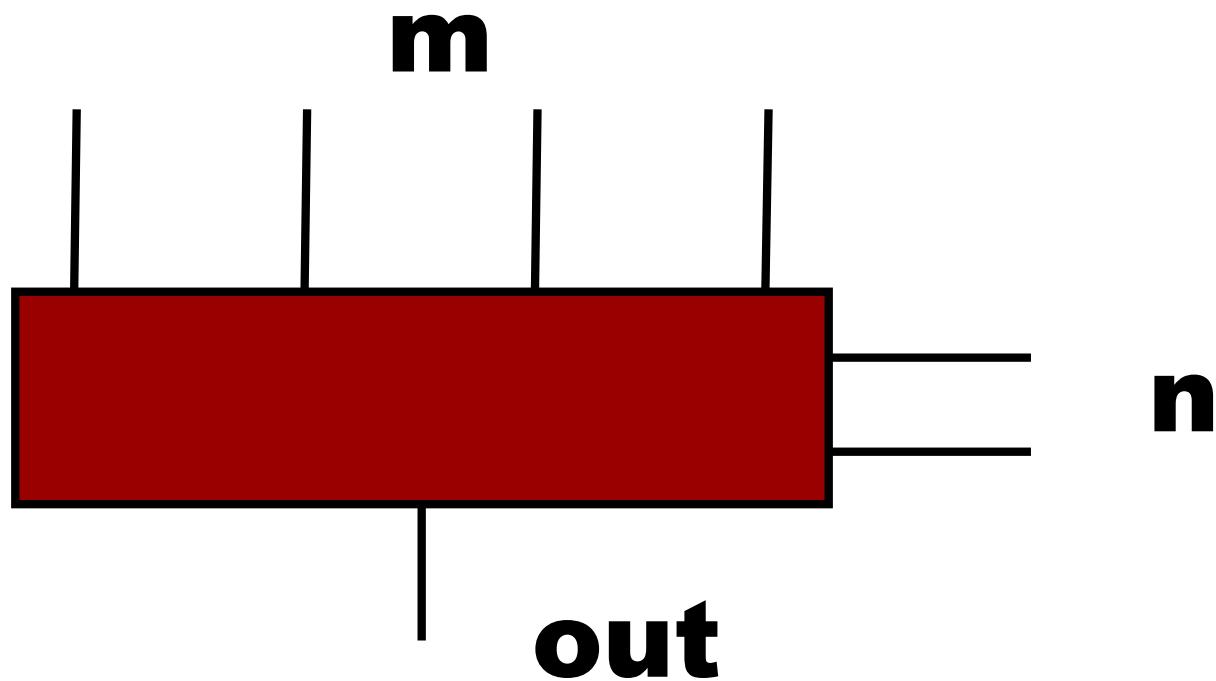
Question

If you have a decoder with n input bits (i.e. an n bit binary number) what is the most outputs the decoder could have?

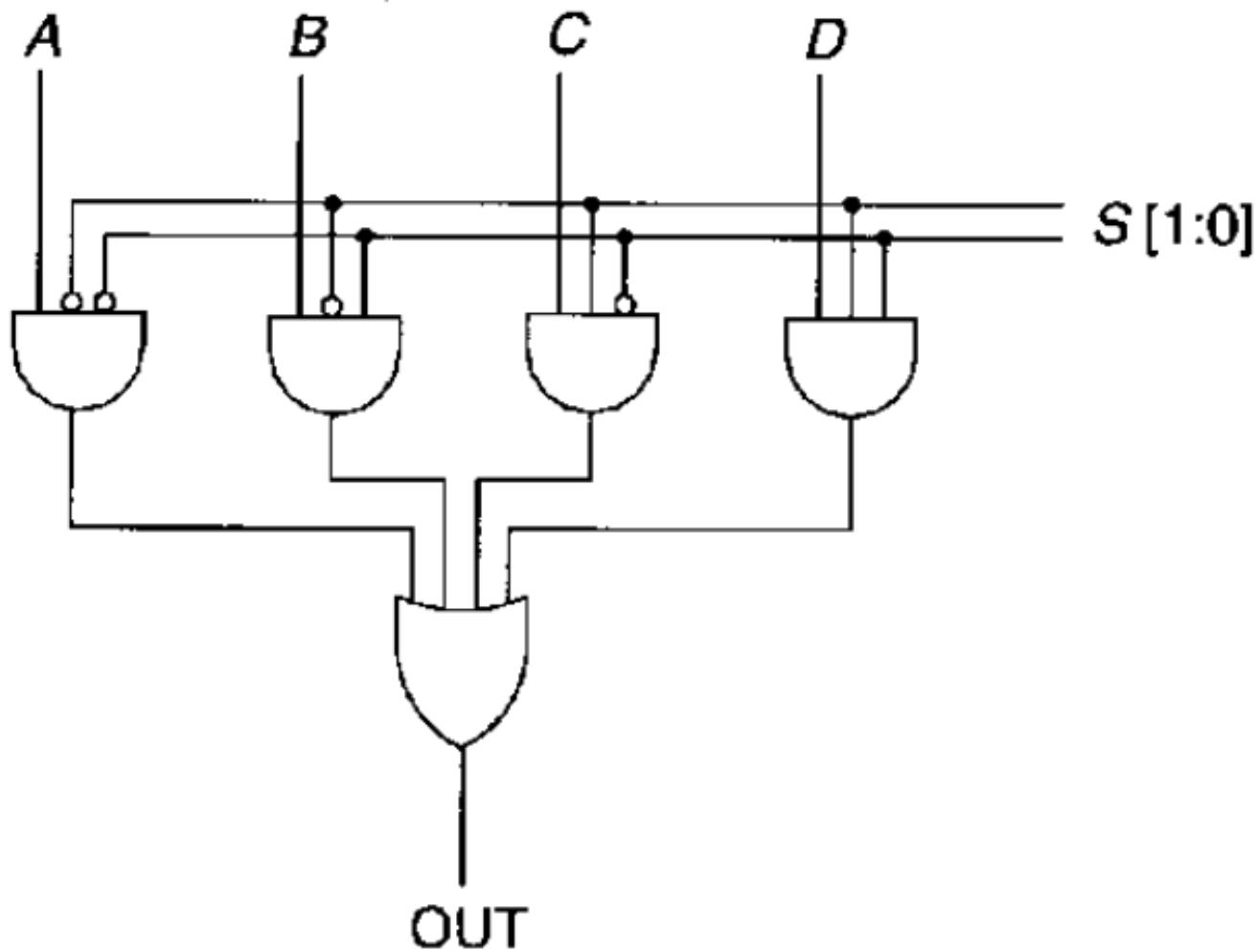
- A. $2n$
- B. n^2
- C. 2^n 
- D. Cannot determine
- E. 4

Problem

- You have m signals and you want to select the logical value on one of them, determined by a set of n control wires



Multiplexor (MUX)



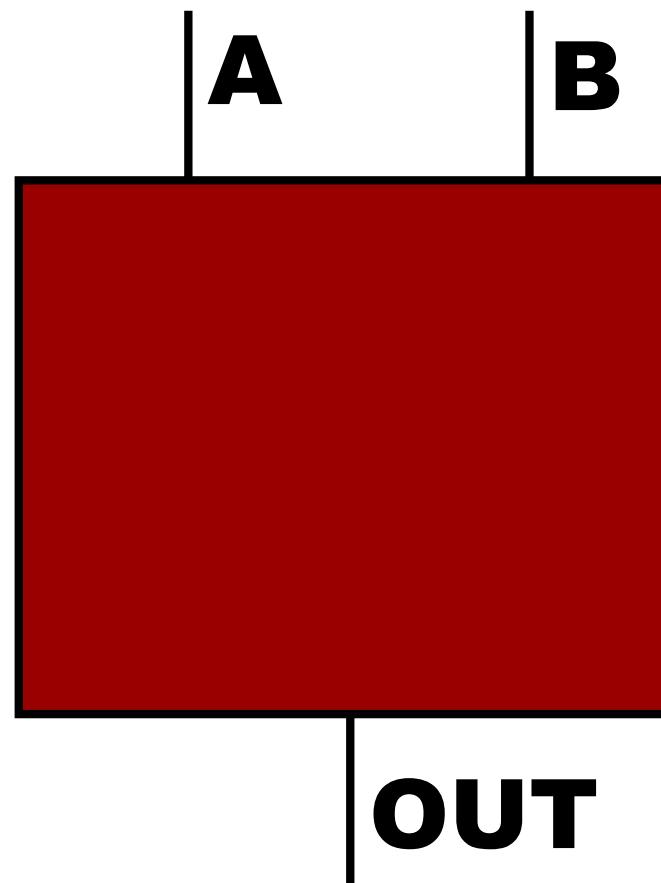
The basic multiplexor has

- A. n outputs, n control lines, 2^n inputs
- B. 1 output, $2n$ control lines, n inputs
- C. 1 output, n control lines, 2^n inputs 
- D. 2^n outputs, 2 control lines, n inputs
- E. 1 output, 4 control lines, 2 inputs

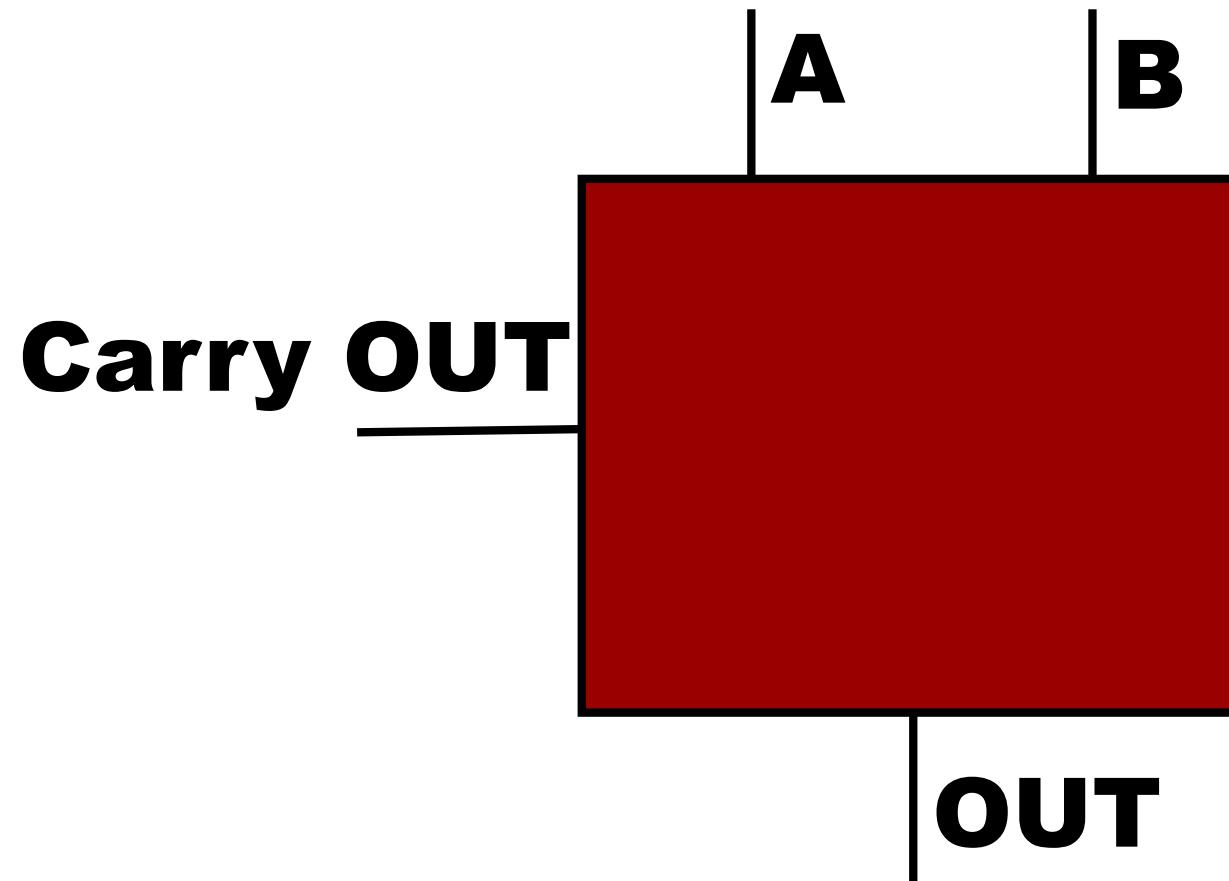
Problem

- ↗ No one will buy your new computer design unless it can do at least some math, say, like adding!

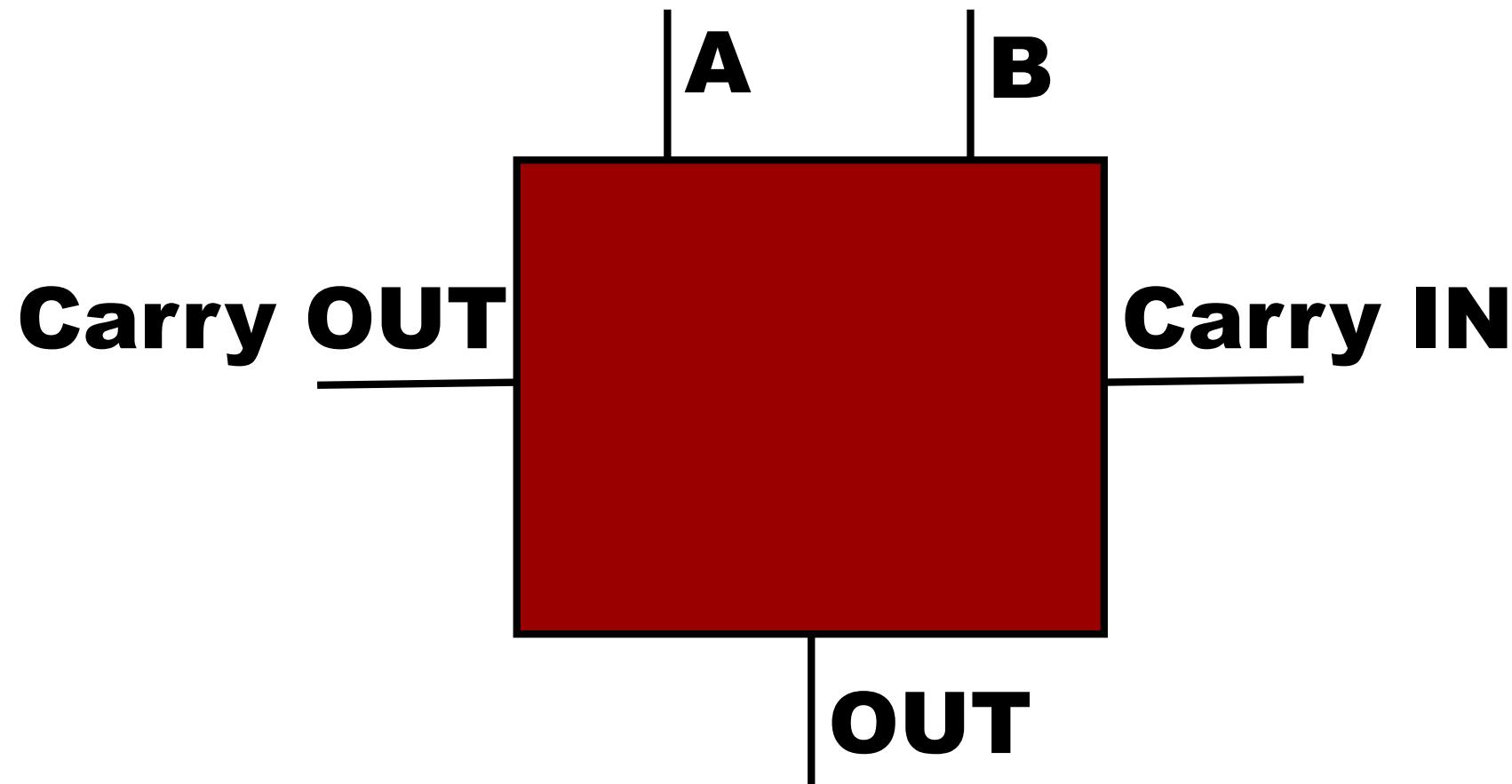
Simple Adder



Half Adder



Full Adder



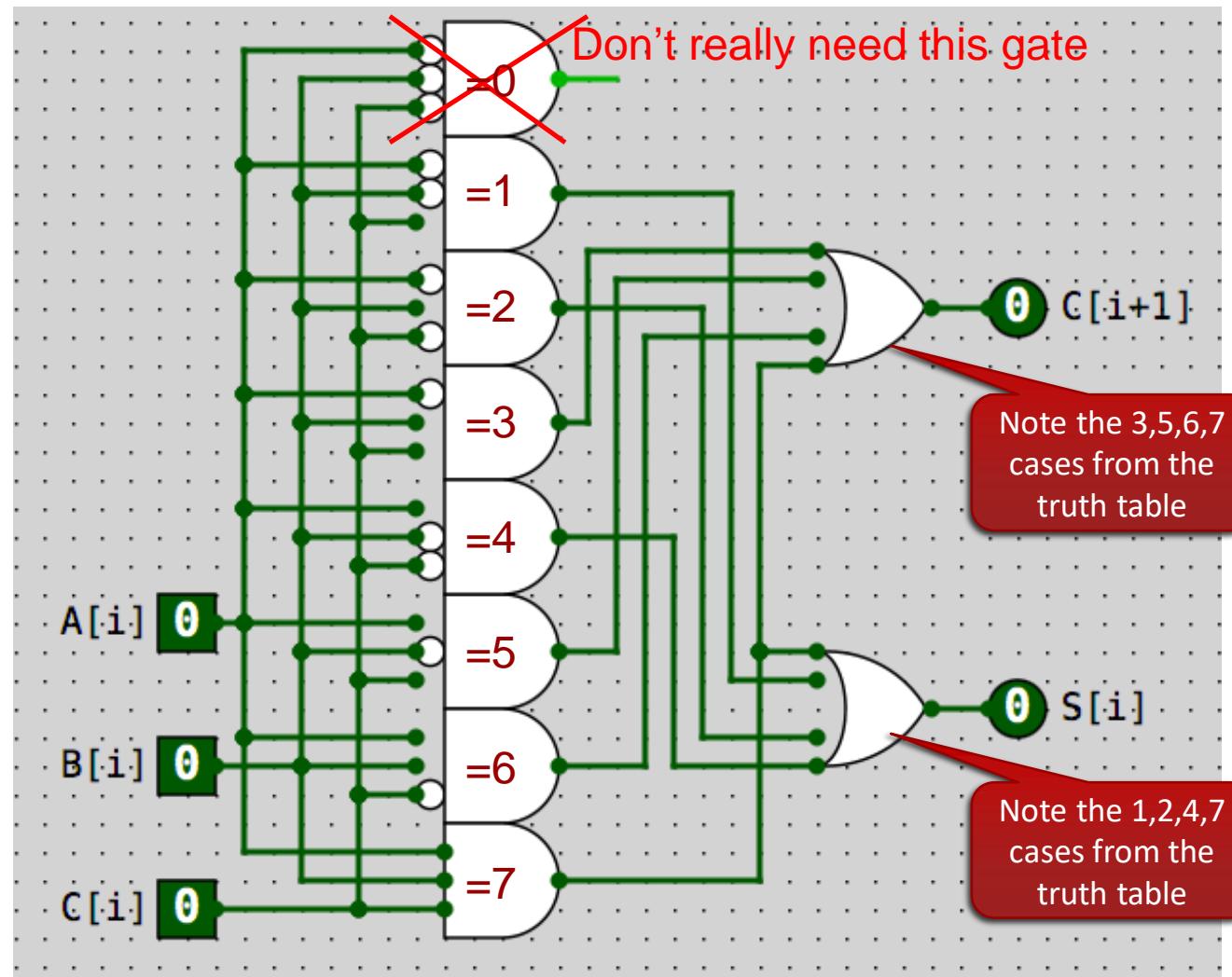
Truth in Adding

A	B	Carry In	Out	Carry Out
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Truth in Adding

A	B	Carry In	Out	Carry Out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Full Adder



The No Thinking Method!

Go buy a full-adder IC chip from Digi-Key.

Boolean Simplification

Boolean Simplification

Basic Boolean algebraic properties

Additive

$$A + B = B + A$$

$$A + (B + C) = (A + B) + C$$

$$A(B + C) = AB + AC$$

Multiplicative

$$AB = BA$$

$$A(BC) = (AB)C$$

$$\overline{\overline{A}} = A$$

Basic Boolean algebraic identities

Additive

$$A + 0 = A$$

$$A + 1 = 1$$

$$A + A = A$$

$$A + \overline{A} = 1$$

Multiplicative

$$0A = 0$$

$$1A = A$$

$$AA = A$$

$$A\overline{A} = 0$$

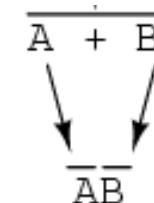
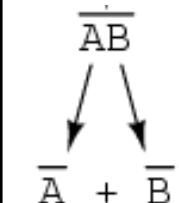
Useful Boolean rules for simplification

$$A + AB = A$$

$$A + \overline{A}B = A + B$$

$$(A + B)(A + C) = A + BC$$

DeMorgan's Laws



Simplify the following Boolean expression:

$$E = ABCD + BC + A'BC$$

$$(E = A \& B \& C \& D \mid B \& C \mid \sim A \& B \& C)$$



- A. BC
- B. A
- C. A'C
- D. AD
- E. D

$$\begin{aligned} E &= ABCD + BC + A'BC \\ &\quad [\text{use identity: } a + ab = a] \end{aligned}$$

$$ABCD + BC = BC$$

$$BC + A'BC = BC$$

$$E = BC$$



A	B	AB	A'	AB+A'	B+A'
0	0	0	1	1	1
0	1	0	1	1	1
1	0	0	0	0	0
1	1	1	0	1	1

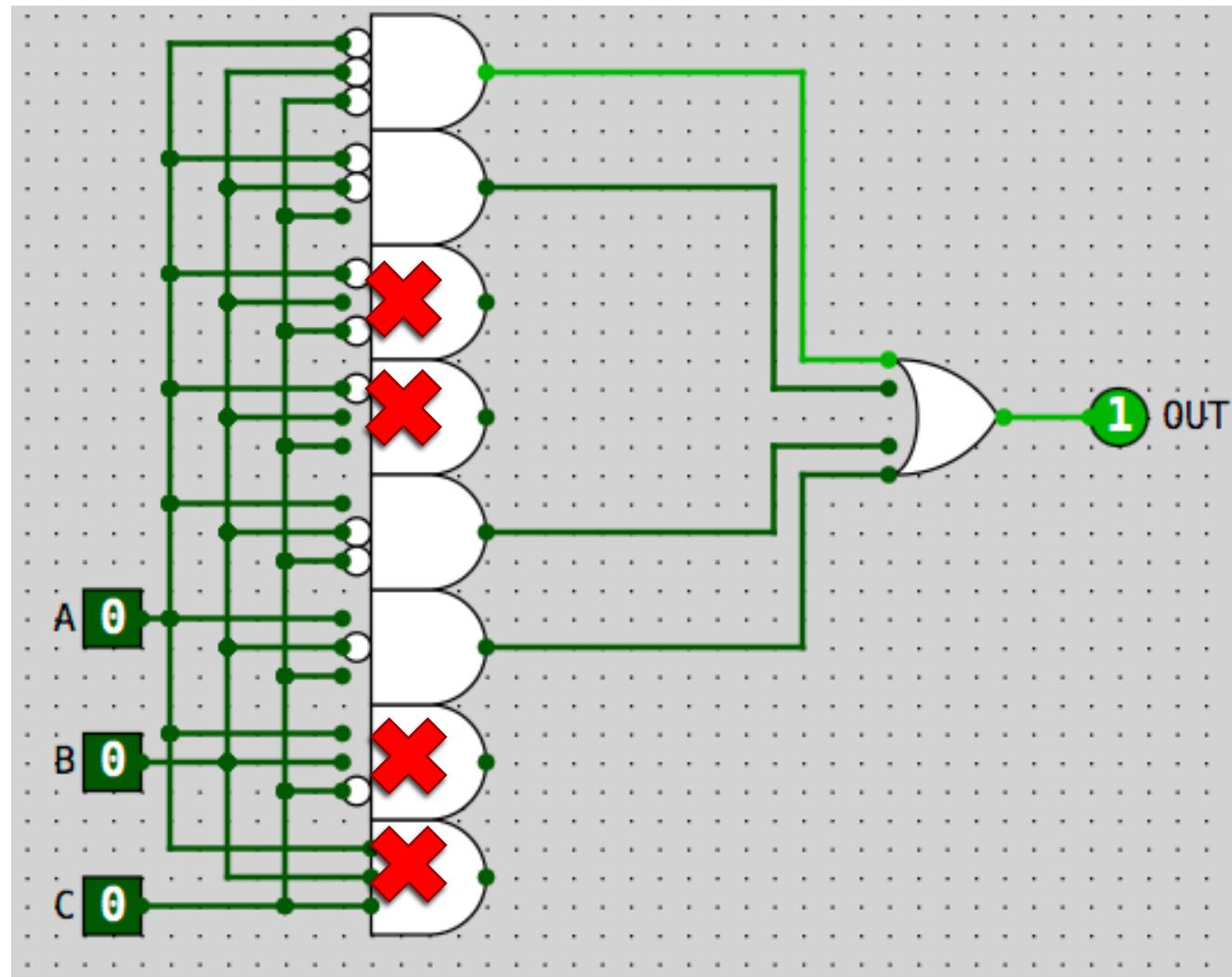
Consider This Circuit

A	B	C	OUT
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Truth Table to Circuit

- ↗ An AND gate for every line in the truth table (i.e. build a decoder) connected to the inputs
- ↗ An OR gate connecting the AND gates for the lines that have 1 in their output column

Implement the Truth Table



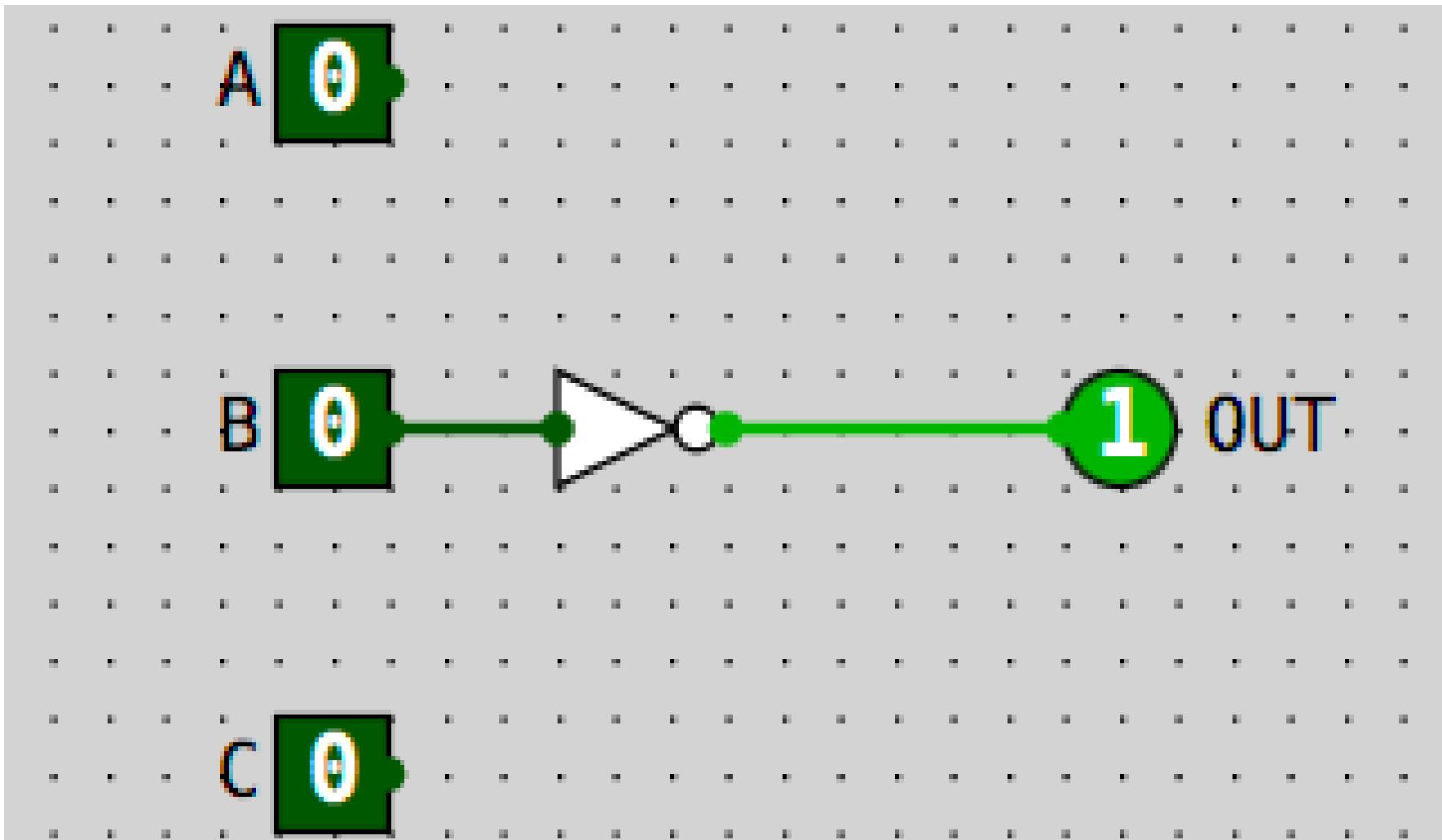
Circuit to Boolean Expression / Truth Table

- ↗ Circuit to Boolean Expression
- ↗ If it's a sum-of-products circuit then
 - ↗ Each AND gate generates a term with the input conditions that yield a 1 from the AND gate (times)
 - ↗ All of the terms are ORed together (plus)
- ↗ Circuit to Truth Table
 - ↗ Each AND gate represents one row of your truth table (inputs)
 - ↗ Each OR gate represents one output column of your truth table
 - ↗ All of the inputs that produce a 1 output are tied to that OR gate

Classic Simplification

- ↗ $F = A'B'C' + A'B'C + AB'C' + AB'C$
- ↗ $F = A'B'(C'+C) + AB'(C'+C)$
- ↗ $F = A'B' + AB'$
- ↗ $F = B'(A'+A)$
- ↗ $F = B'$

Quite a Simplification, No?



Something to Note

- We are used to writing sum-of-products with algebraic polynomials, e.g.

$$X^3 + 4X^2 + 2X + 12$$

- Turns out that truth tables (and other forms) naturally yield boolean expressions in sum-of-products form, e.g.

$$ABC + ABC' + A'BC + A'B'C'$$

- This turns out to be very convenient for implementing circuits

- ↗ Boolean Expression
- ↗ Truth Table
- ↗ Combinational Circuit
- ↗ Karnaugh Map

- ↗ You can change between ANY of these forms without losing information!

- ↗ Resources:

- ↗ <http://electronics-course.com/karnaugh-map>
- ↗ <http://www.32x8.com/circuits7>

- ↗ Can you convert
 - ↗ A Karnaugh map to a combinational circuit?
 - ↗ A truth table to a Karnaugh map
 - ↗ A truth table to a combinational circuit?
 - ↗ A Boolean expression into a Karnaugh map?
 - ↗ A Boolean expression into a truth table?
 - ↗ A truth table into a Boolean expression?

Truth Table

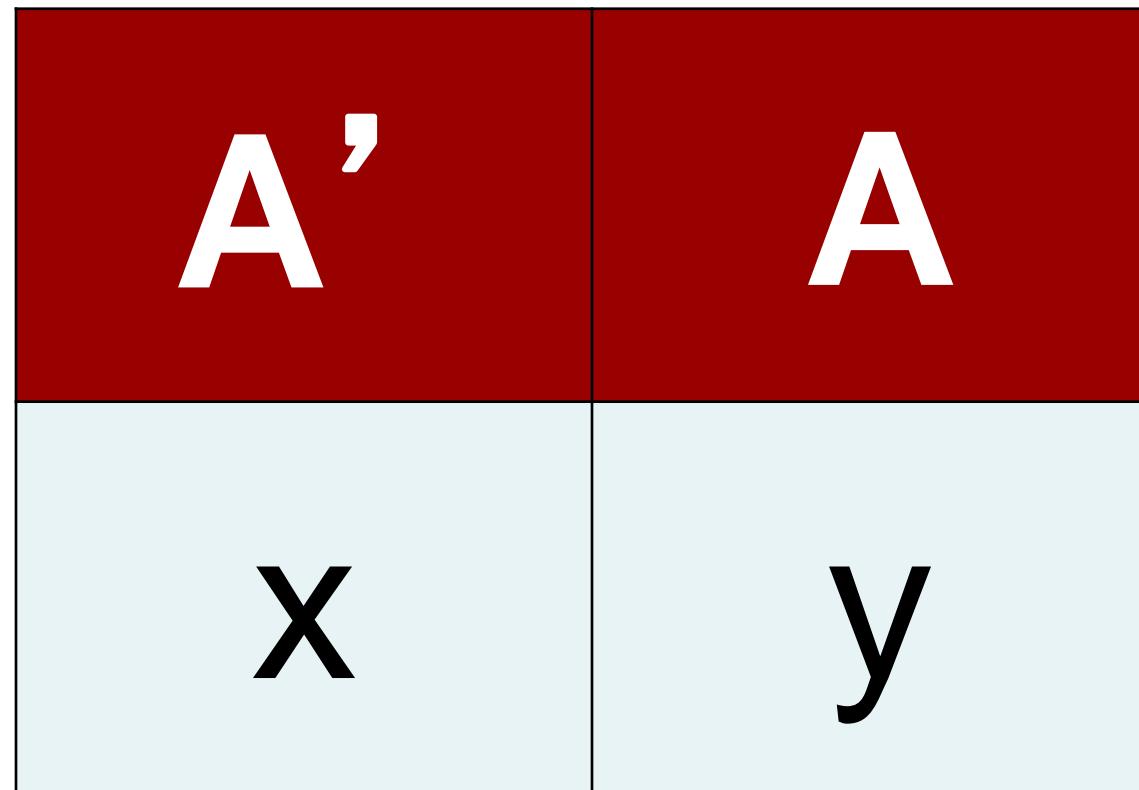
A	OUT
0	x
1	y

1 Variable Karnaugh Map

A'	A
What is the output if A is false?	What is the output if A is true?

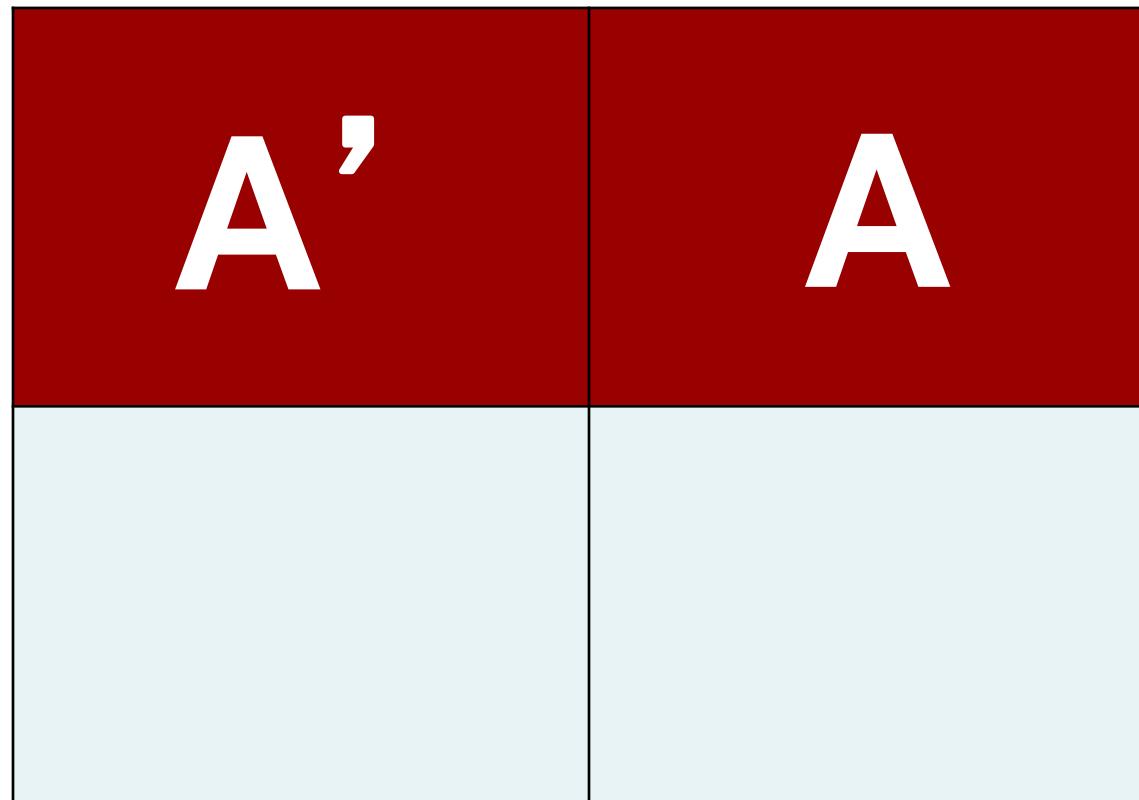
What does it mean if both boxes contain 1?

1 Variable Karnaugh Map



What does it mean if both boxes contain 1?

Map: OUT = A'



Truth Table: OUT = A'

A	OUT
0	1
1	0

Map: OUT = A'

A'	A
1	0

2-Variable Truth Table

A	B	OUT
0	0	x_0
0	1	x_1
1	0	x_2
1	1	x_3

2-Variable K-Map

	B'	B
A'	What is the output if both A and B are false?	What is the output if A is false and B is true?
A	What is the output if A is true and B is false?	What is the output if both A and B are true?

2-Variable K-Map

	B'	B
A'	x_0	x_1
A	x_2	x_3

Map: OUT = A'B' + AB

	B'	B
A'		
A		

Map: OUT = A'B' + AB

	B'	B
A'	1	0
A	0	1

Can this expression be simplified?

Map: OUT = A'B' + AB'

	B'	B
A'		
A		

Map: OUT = A'B' + AB'

	B'	B
A'	1	0
A	1	0

Can this expression be simplified?

Question

Which of these conversions between forms of a circuit are allowed because the forms are equivalent?

- A. A Karnaugh map to a combinational circuit
- B. A truth table to a Karnaugh map
- C. A truth table to a combinational circuit
- D. A Boolean expression into a Karnaugh map
- E. A Boolean expression into a truth table
- F. All of the above

- Suppose you had to give a series of 3 bit codes to disarm the detonation sequence of a nuclear device you were sitting on.
- There would be 8 codes and they had to go in a certain sequence picked by you.
- Any deviation would result in an immediate detonation of the nuclear device
- What codes in what order do you use?

Classic Sequence

000

001

010

011

100

101

110

111

010 } Two switches change at once!
Can you get the timing perfect?
Boom!

Gray Code Sequence

000

001

011

010

110

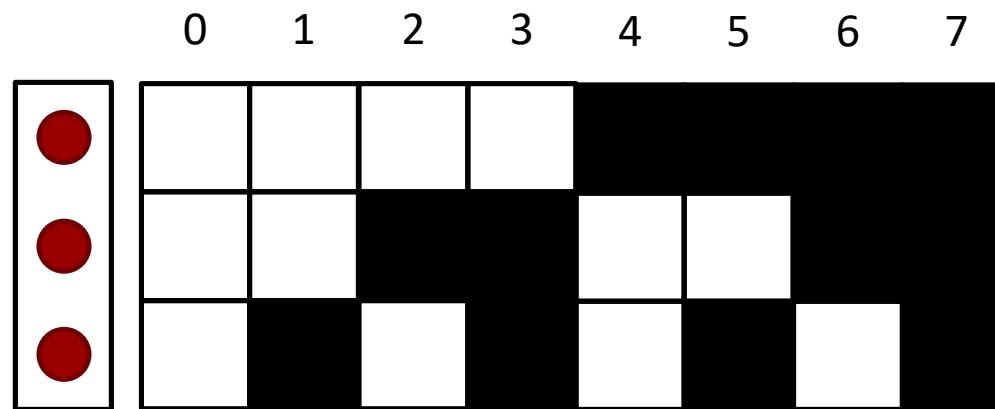
111

101

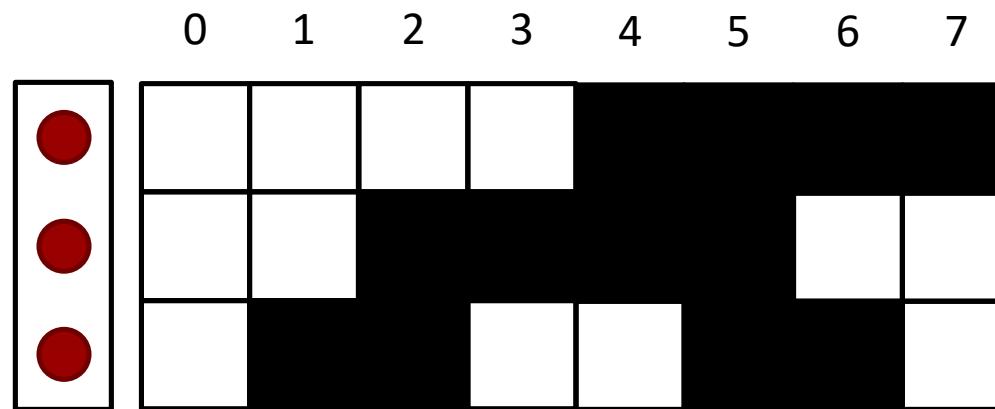
100

Only one switch per transition!
All the way to the end!
No boom!

Optical Encoder

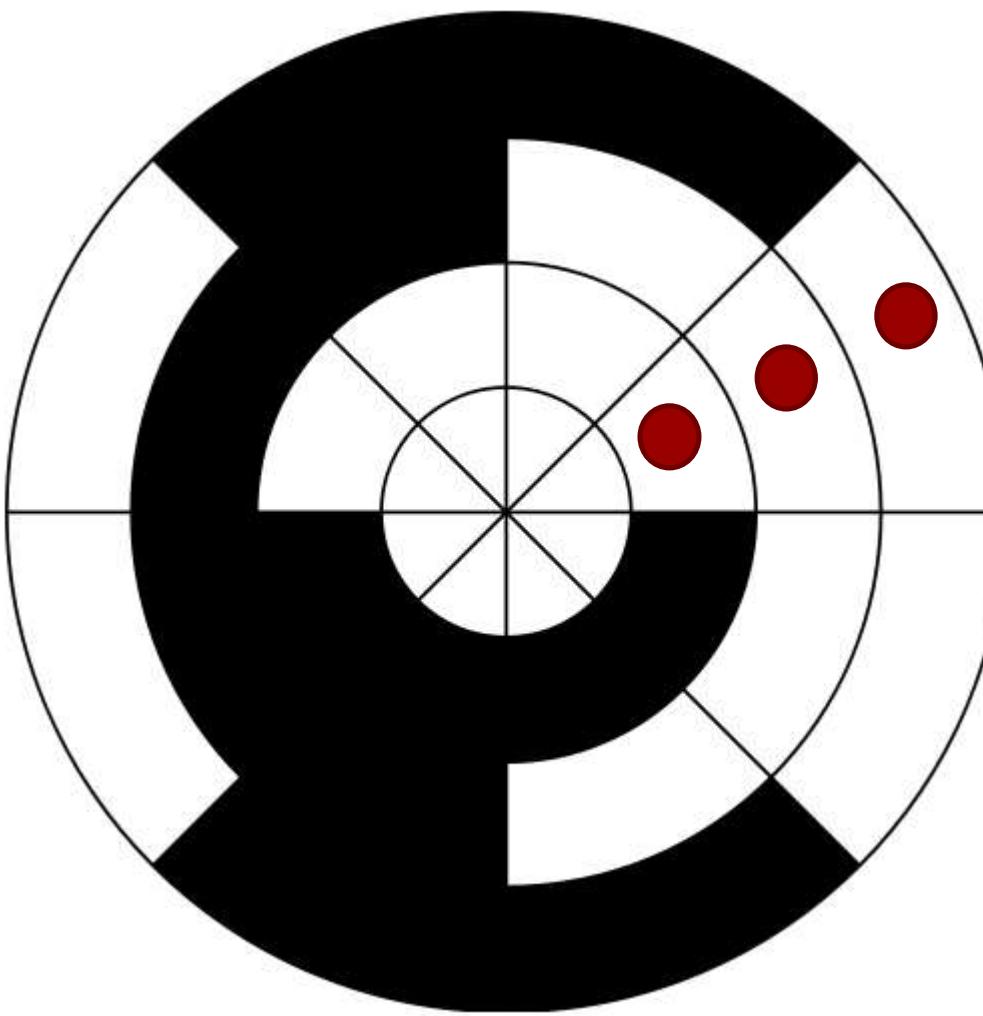


Gray Code Optical Encoder



3-bit Gray Code

000
001
011
010
110
111
101
100



2-bit Gray Code

00
01
11
10

This is the one you need to remember.

4-bit Gray Code

0000
0001
0011
0010
0110
0111
0101
0100
1100
1101
1111
1110
1010
1011
1001
1000

3-Variable Truth Table

A	B	C	OUT
0	0	0	x_0
0	0	1	x_1
0	1	0	x_2
0	1	1	x_3
1	0	0	x_4
1	0	1	x_5
1	1	0	x_6
1	1	1	x_7

Map: $A'B'C + A'BC + ABC$

	$B'C'$ (00)	$B'C$ (01)	BC (11)	BC' (10)
A' (0)	x_0	x_1	x_3	x_2
A (1)	x_4	x_5	x_7	x_6

Map: $A'B'C + A'BC + ABC$

	$B'C'$	$B'C$	BC	BC'
A'	0	1	1	0
A	0	0	1	0

Can this expression be simplified?

Map: $A'B'C + A'BC + ABC$

	$B'C'$	$B'C$	BC	BC'
A'	0	1 1	1	0
A	0	0	1	0

Can this expression be simplified?

How Does the Map Remove Variables?

- ↗ $A'B'C + A'BC + ABC$
- ↗ Two adjacent 1s in the map means there is an $x + x'$ in the formula; the map tells us where
- ↗ $A'C(B' + B) + ABC$
- ↗ $A'C + ABC$

Map: $A'C + ABC$

	$B'C'$	$B'C$	BC	BC'
A'	0	1	1	0
A	0	0	1	0

Can this expression be simplified: $A'C + BC$

How Does the Map Remove Variables?

- ↗ $A'B'C + A'BC + ABC$
- ↗ Two adjacent 1s in the map means there is an $x + x'$ in the formula; the map tells us where
- ↗ $A'C(B' + B) + ABC$
- ↗ $A'C + ABC$
- ↗ $C(A' + AB)$
- ↗ $C(A' + B)$
- ↗ $A'C + BC$

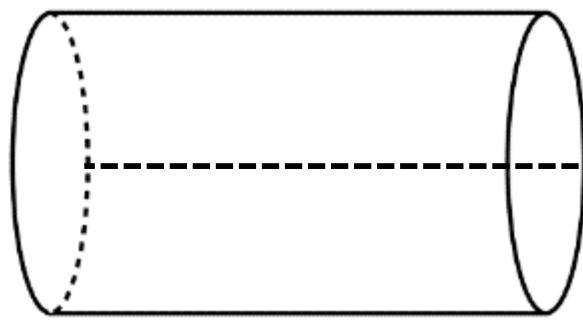
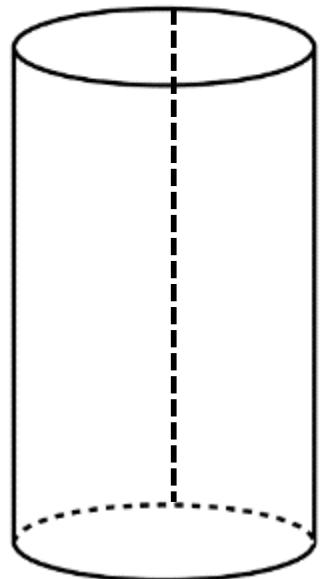
Map: $A'B'C' + ABC' + AB'C' + A'BC'$

	$B'C'$	$B'C$	BC	BC'
A'	X_0	X_1	X_3	X_2
A	X_4	X_5	X_7	X_6

Map: $A'B'C' + ABC' + AB'C' + A'BC'$

	$B'C'$	$B'C$	BC	BC'
A'	1	0	0	1
A	1	0	0	1

Can this expression be simplified?



Wrap!

Map: $A'B'C' + ABC' + AB'C' + A'BC'$

	$B'C'$	$B'C$	BC	BC'
A'	1	0	0	1
A	1	0	0	1

Expression simplifies to C'

4-Variable Truth Table

A	B	C	D	OUT
0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

	CD	00	01	11	10
A	00	0	1	3	2
B	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

4-Variable K-Map

	$C'D'$	$C'D$	CD	CD'
$A'B'$				
$A'B$				
AB				
AB'				

4-Variable Truth Table

A	B	C	D	OUT
0	0	0	0	1
0	0	0	1	0
0	0	1	0	1
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

	CD				
A	00	01	11	10	
B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Map: $A'B'C'D' + AB'CD' + AB'C'D' + A'B'CD'$

	$C'D'$	$C'D$	CD	CD'
$A'B'$				
$A'B$				
AB				
AB'				

	CD	00	01	11	10
A B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Map: $A'B'C'D' + AB'CD' + AB'C'D' + A'B'CD'$

	C' D'	C' D	CD	CD'
A' B'	1			1
A' B				
AB				
AB'	1			1

Can this expression be simplified?

	CD	00	01	11	10
A B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Map: $A'B'C'D' + AB'CD' + AB'C'D' + A'B'CD'$

	$C'D'$	$C'D$	CD	CD'
$A'B'$	1			1
$A'B$				
AB				
AB'	1			1

Simplifies to $B'D'$

	CD	00	01	11	10
A	00	0	1	3	2
B	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Each Pair Eliminates 1 Variable

	C' D'	C' D	CD	CD'
A' B'				
A' B	1			1
AB		1		
AB'		1		

Simplifies to A'BD' + AC'D

	CD				
A	00	01	11	10	
B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Eliminates 2 Variables

	C' D'	C' D	CD	CD'
A' B'				
A' B				
AB				
AB'				
1				
1				

Simplifies to AD

	CD	00	01	11	10
A B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Eliminates 3 Variables

	C' D'	C' D	CD	CD'
A' B'	1			1
A' B	1			1
AB	1			1
AB'	1			1

Simplifies to D'

	CD				
A	00	01	11	10	
B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Eliminates 3 Variables

	C' D'	C' D	CD	CD'
A' B'				
A' B	1	1	1	1
AB	1	1	1	1
AB'				

Simplifies to B

	CD	00	01	11	10
A B	00	0	1	3	2
	01	4	5	7	6
	11	12	13	15	14
	10	8	9	11	10

Don't care: Eliminates 3 Variables

	$C' D'$	$C' D$	CD	CD'
$A' B'$				
$A' B$	1	1	1	1
AB	X	1	1	1
AB'	X			

Five Variable

E'	C' D'	C' D	CD	CD'
A' B'				
A' B				
AB				
AB'				

E	C' D'	C' D	CD	CD'
A' B'				
A' B				
AB				
AB'				

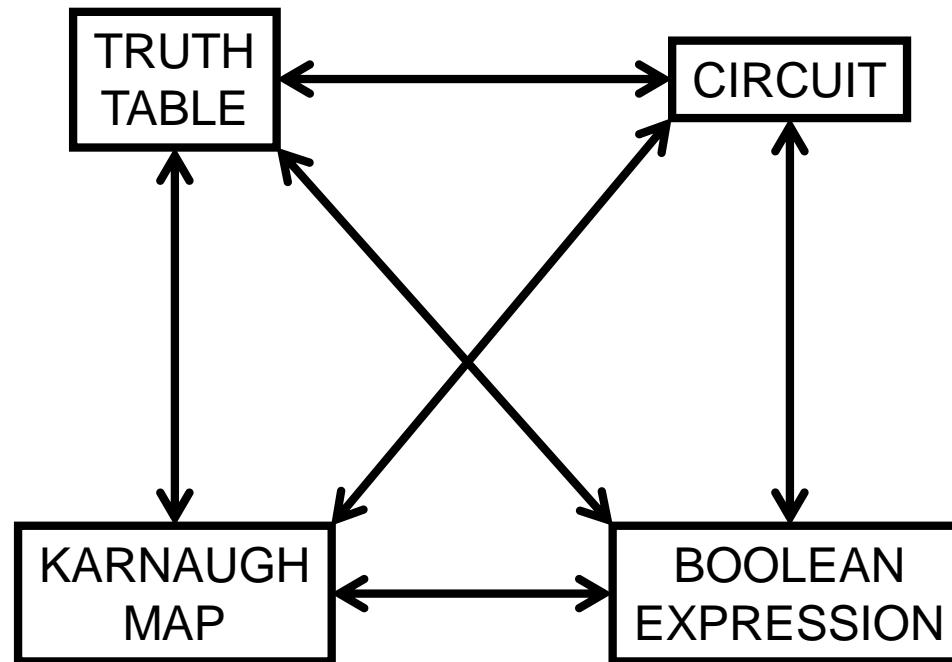
6 Variable

E'		F'					F				
E		C' D'	C' D	CD	CD'	A' B'	C' D'	C' D	CD	CD'	
A' B'						A' B'					
A' B						A' B					
AB						AB					
AB'						AB'					
E'		C' D'	C' D	CD	CD'	A' B'	C' D'	C' D	CD	CD'	
A' B'						A' B'					
A' B						A' B					
AB						AB					
AB'						AB'					

Remember: Same Data, Different Form

- Combinational Logic Circuit
- Boolean Expression
- Truth Table
- Karnaugh Map

Can You Convert Any of These to Any Other?



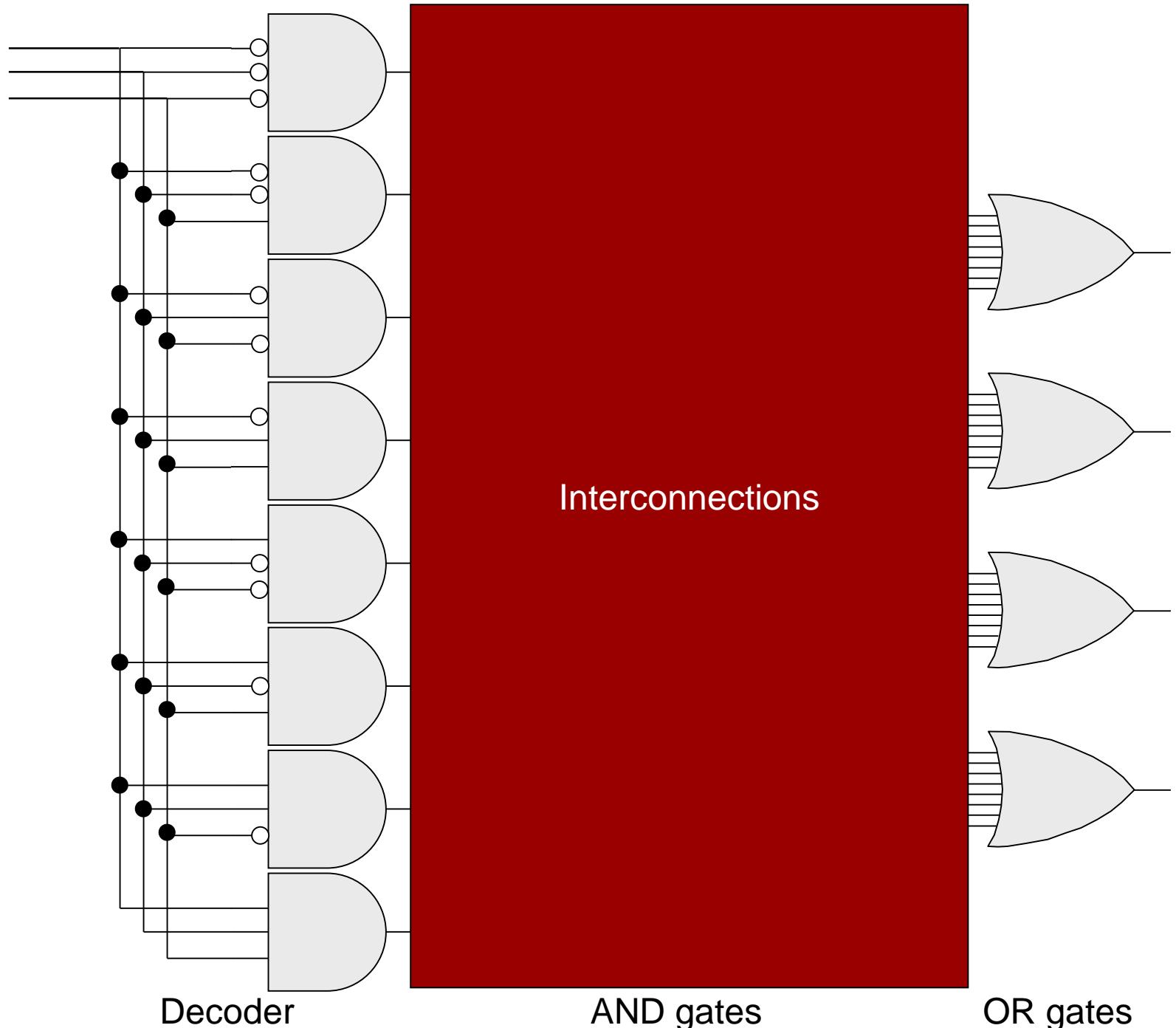
Strategy for 4-variable K-Maps

- ↗ Find a box around 16 1s.
 - ↗ If so, you are done.
- ↗ Find boxes around groups of 8 1s.
 - ↗ If no 1s are left, you are done.
- ↗ Find boxes around groups of 4 1s.
 - ↗ If no 1s are left, you are done.
- ↗ Find boxes around groups of 2 1s.
 - ↗ If no 1s are left, you are done.
- ↗ Find boxes around groups of single 1s.
- ↗ Done: Now write a term for every box you drew.

Some Automation

- ↗ Try <http://32x8.com>
- ↗ Choose “4 variables” and “Kmap with Don’t cares” input option
- ↗ Watch it draw the circuits directly from the Kmaps

- Given a truth table we can implement it using output using a series of NOT gates, AND gates and then OR gates (remember sum-of-products?)
- We need 2^n AND gates where n is the number of inputs
 - How many rows does the truth table have?
- We need one OR gate for each output
- General purpose ***Programmable Logic Array or (Field) Programmable Gate Array*** devices exist for this purpose



PLA/PGA

- ↗ Transistors
- ↗ Logic Gates
 - ↗ NOT, OR, NOR, AND, NAND
 - ↗ DeMorgan's Law
 - ↗ Larger Gates
- ↗ Combinational Logic Circuits
 - ↗ Decoder, MUX, Full Adder, PLA,
 - ↗ Logical Completeness
- ↗ Simplification
 - ↗ Boolean
 - ↗ Karnaugh Maps
 - ↗ PLA/PGA

Digital Logic II



So Where Are We Going With This?

- ↗ Sequential Logic Circuits
 - ↗ State
 - ↗ Memory
 - ↗ Address space
 - ↗ Addressability
 - ↗ $2^2 \times 3$ bit memory
 - ↗ Clocks
 - ↗ Edge-triggered and level-triggered flip-flops
 - ↗ Example State Machine
 - ↗ Design
 - ↗ Simplification
 - ↗ Implementation
 - ↗ One-hot and Encoded State Machines

Warning

- ↗ Another big concept approaching!

Combinational vs. Sequential Logic

↗ Combinational

- ↗ Combination of AND, OR, NOT (plus NAND & NOR)
- ↗ Same inputs always produce same output
- ↗ Analogous to cheap bicycle lock

↗ Sequential

- ↗ Requires **storage elements**
- ↗ Output depends on inputs plus **state**
- ↗ Analogous to a RLR combination lock
- ↗ Used to build memory & ***state machines***

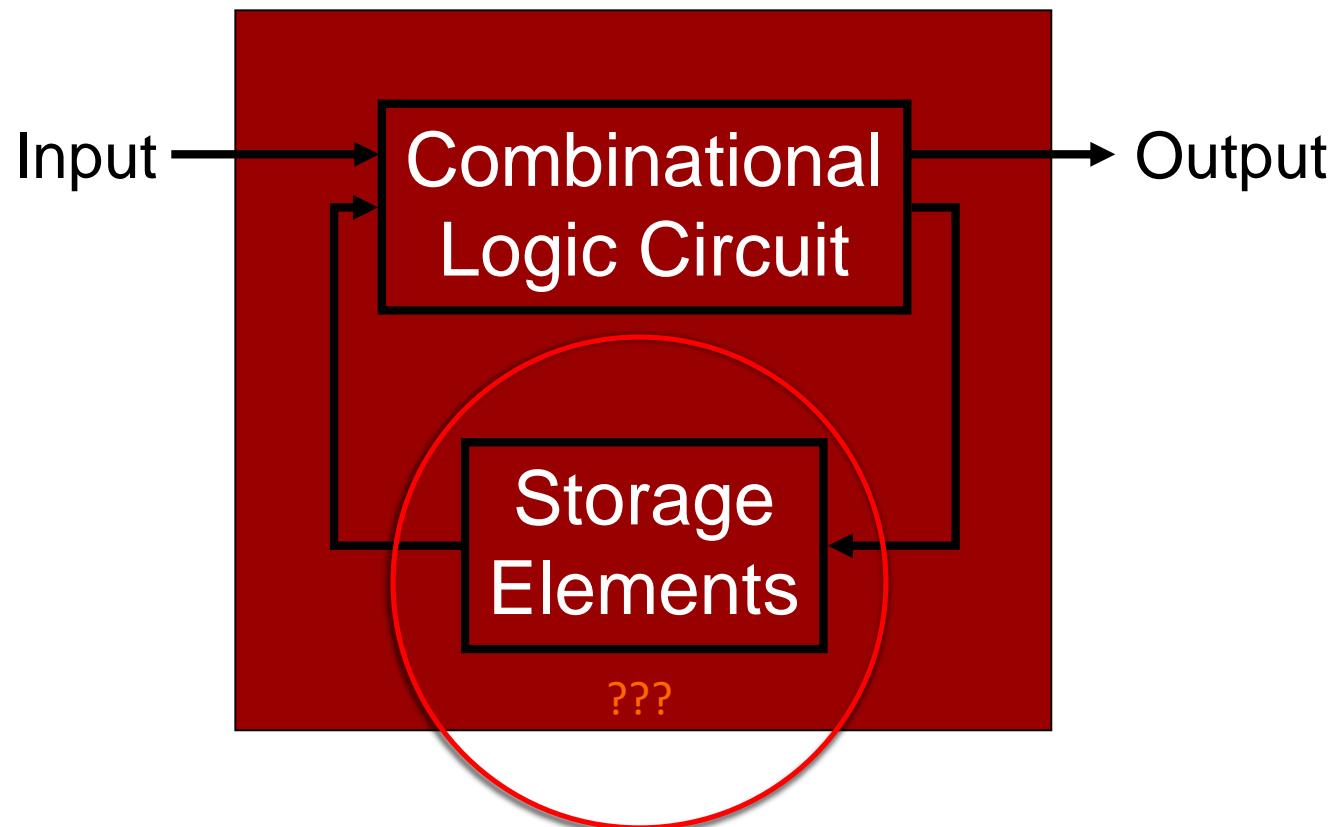
So What is State?



State or No State?

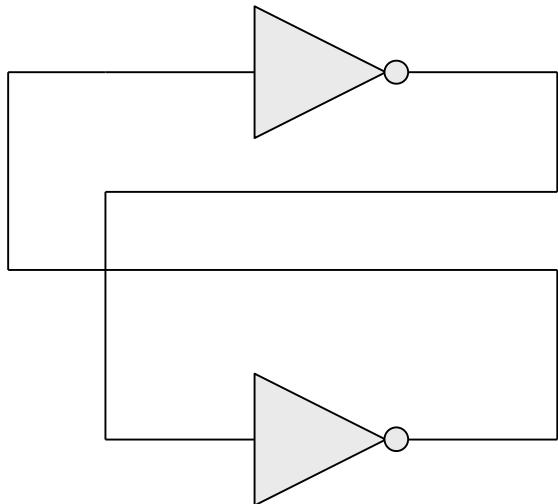


Sequential Logic Circuit

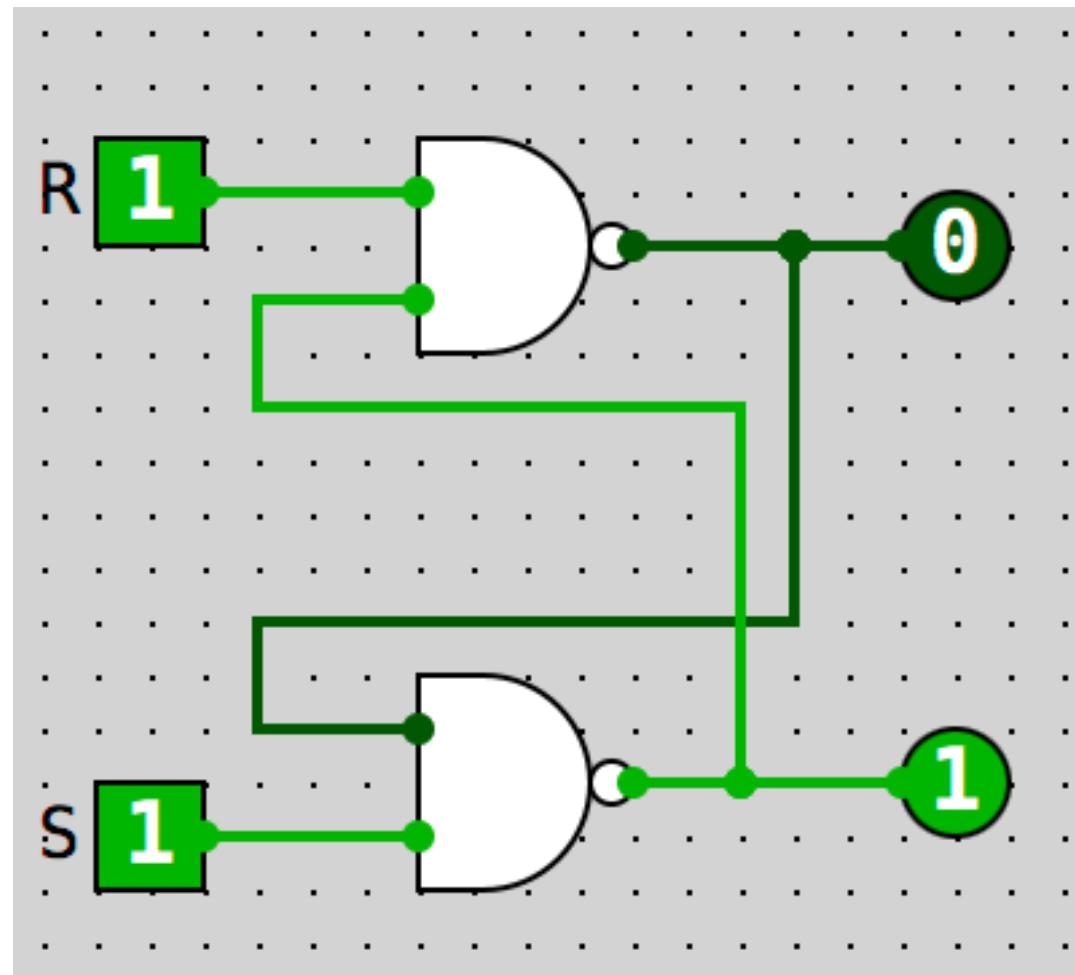


Basic Storage

Consider:



R/S Latch



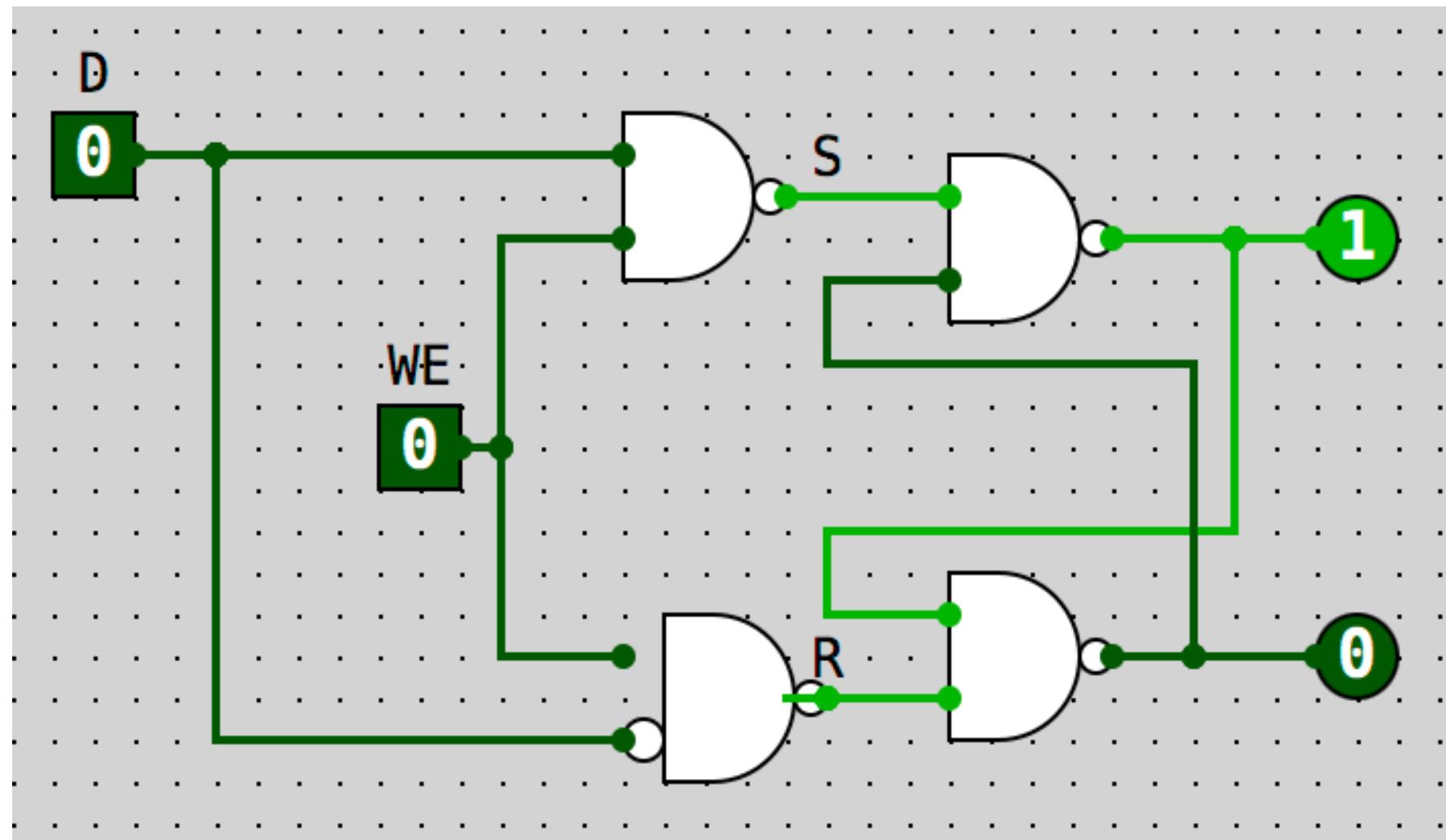
What happens if both inputs to the R/S latch are 0?

Gated D Latch

- ☞ The RS Latch can be set or cleared but what if we want to record the value presented on the input?

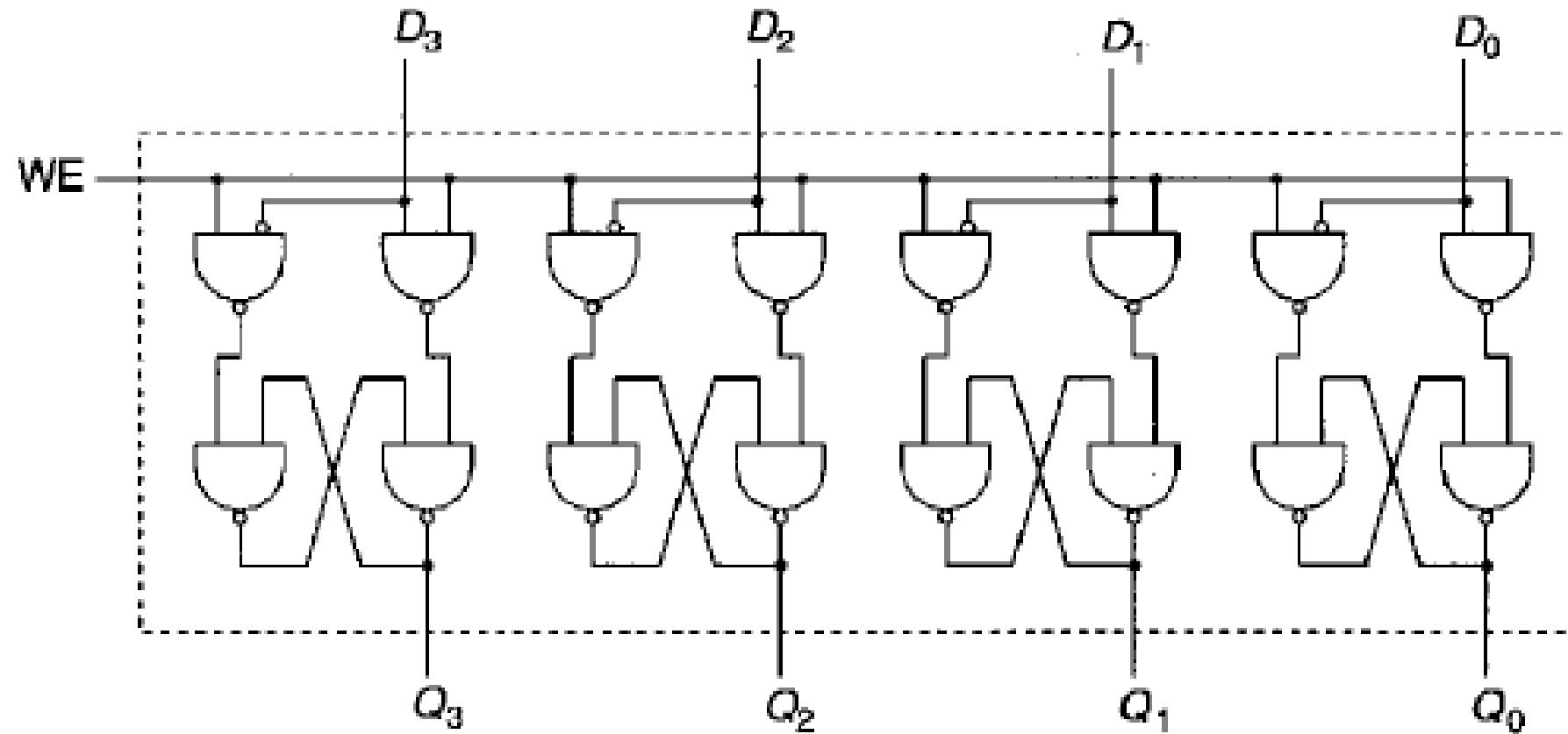


Gated D Latch



What is a Register?

- Basically an array of Gated D latches!



Memories (or How is This Different From Registers?)

- ↗ Do you think it would be useful to have more than one storage location accessible with only one set of wires?

Definitions

- Address Space -- How many addresses are possible?
- Addressability -- How big is each memory location?

Question

If you have a memory with a 16-bit address in which each byte is given a distinct memory address, what is the addressability of this memory?

- A. 8 bits
- B. 16 bits
- C. 24 bits
- D. 32 bits



Question

If you have a memory with a 16-bit address in which each byte is given a distinct memory address, what is this memory's address space?

- A. 256
- B. 65536
- C. 524288
- D. 4294967296



- Basic Building Blocks of a Memory

- Register
 - Gated D Latch
- Decoder
- Multiplexors

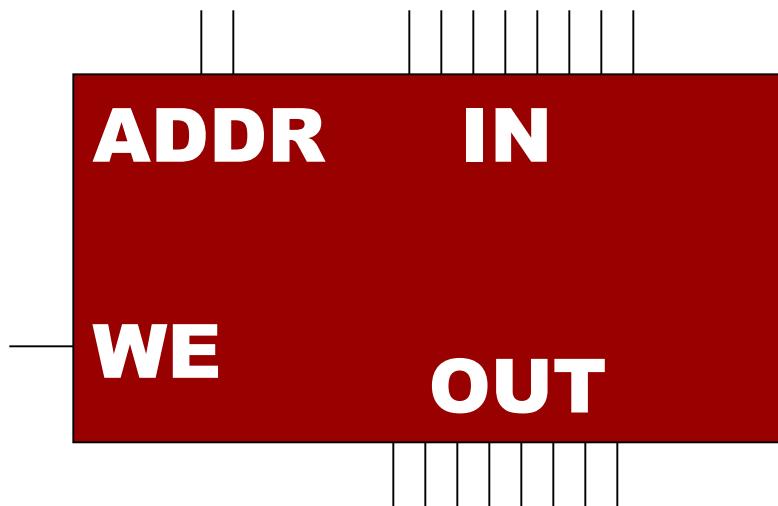
Memory

- ↗ Looking from the outside, what do we need?

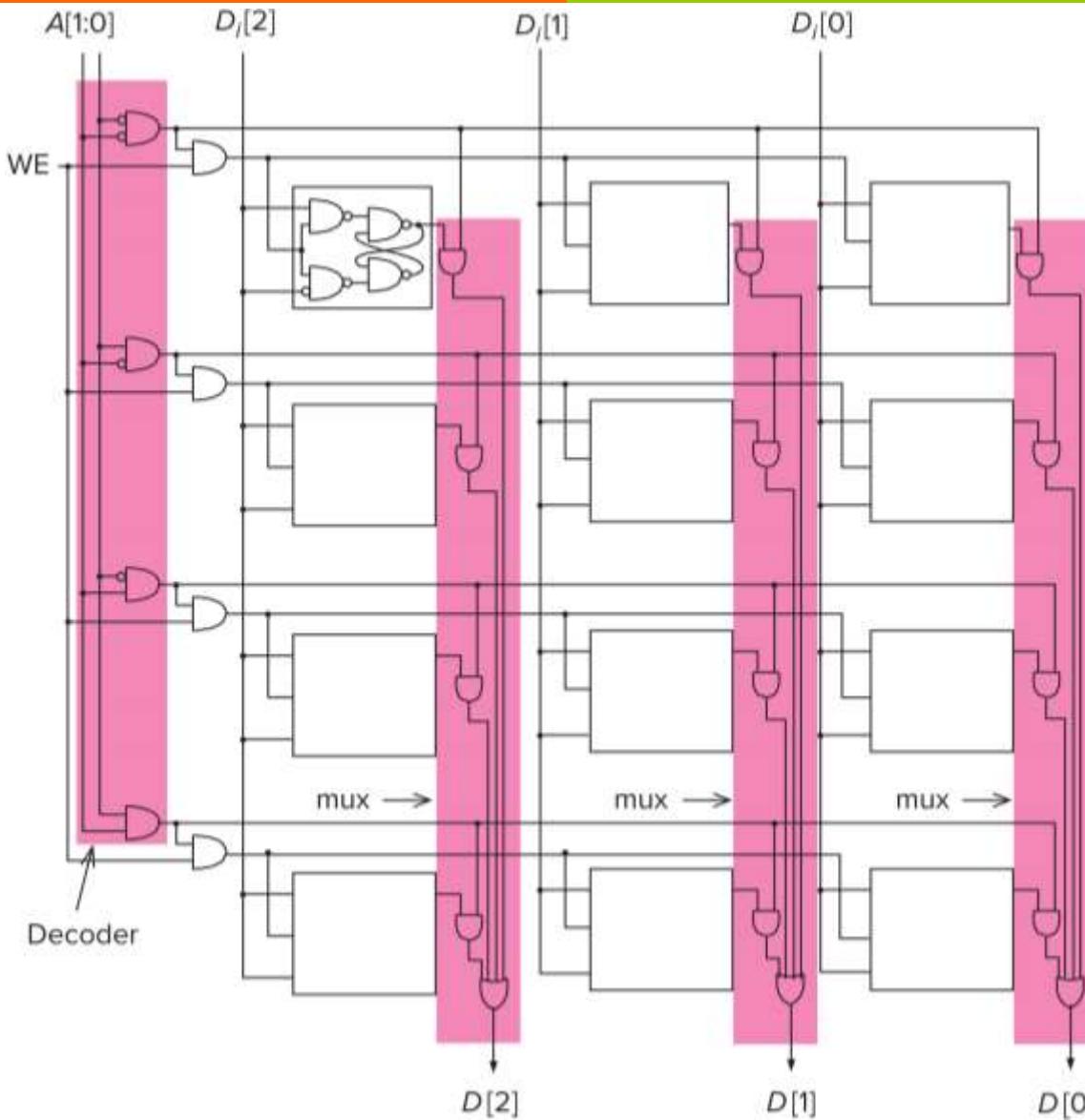


Memory

- Looking from the outside, what do we need?



A $2^2 \times 3$ -bit Word Memory

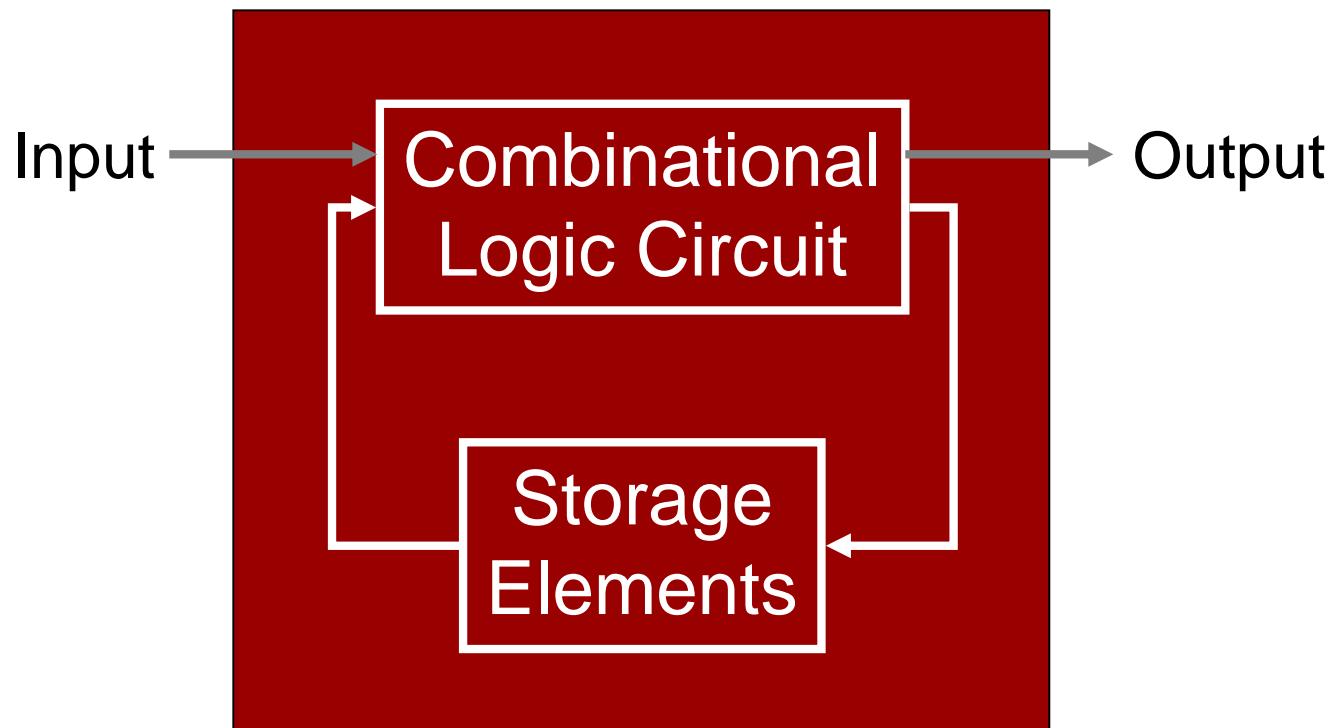


Sequential Logic Circuits, Now That We Have Storage

Remember: Compare and Contrast

- Combinational Logic Circuits
 - Make decisions
 - Same inputs always produce same output
 - Depends on what is **happening now**
- Sequential Logic Circuits
 - Make decisions and store information
 - Output depends on **inputs AND state**
 - Depends on what has **happened in the past** as well as what is **happening now**

Sequential Logic Circuit



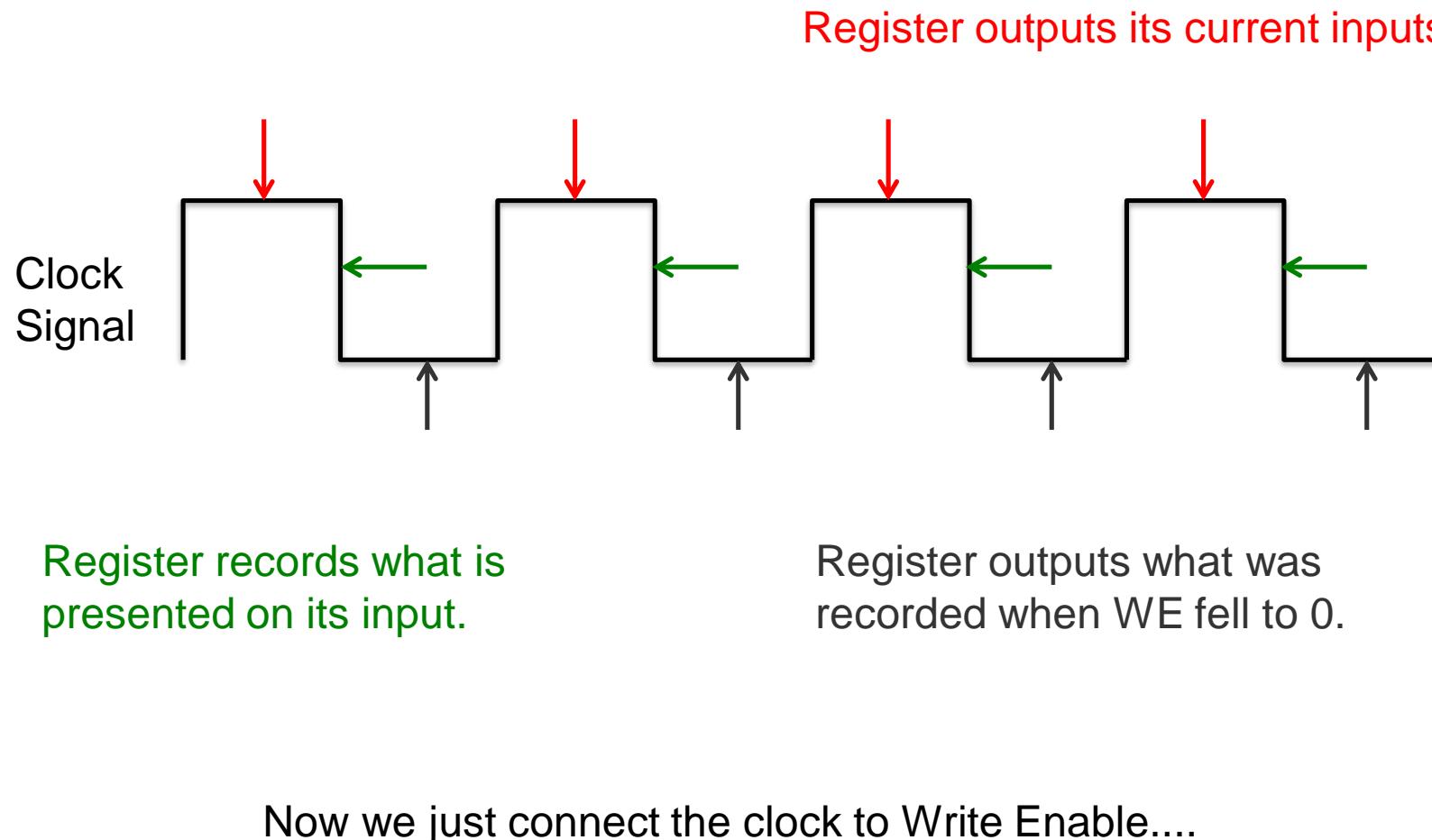
A Wrinkle With the D Latch

- ↗ Consider the Gated D latch...
- ↗ It is simple, and simple is what you want if you are going to implement a memory using 4 billion of them
- ↗ But there is a drawback...
- ↗ A Gated D Latch is a *level-triggered* device, that is, it will store whatever value is present on the input when the write enable goes from true to false

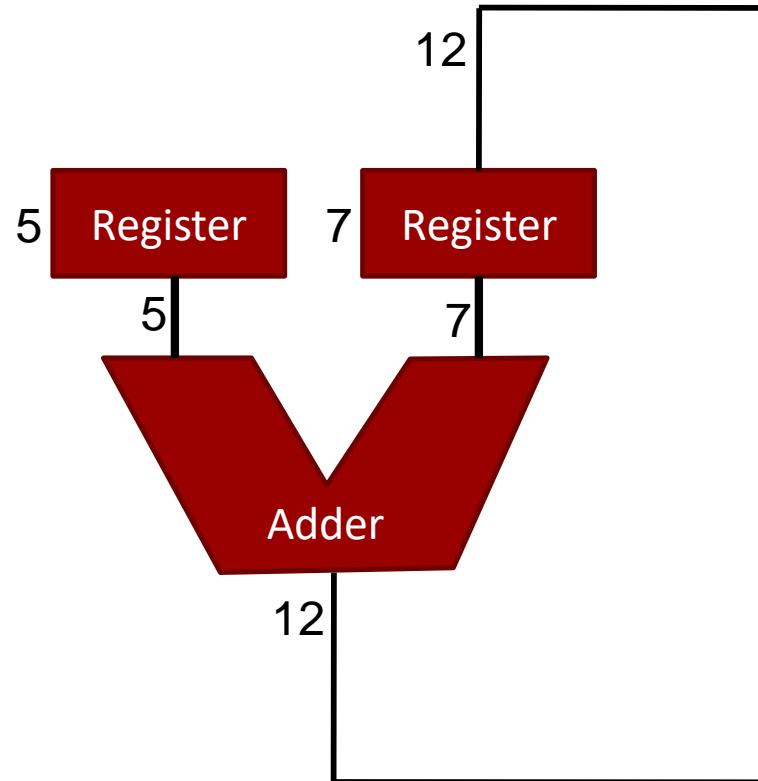
Clocked Logic

- To make the job of designing digital circuits easier it is useful to synchronize the operation using a clock
- But there's an issue when *outputs are used as inputs* to the same circuit element....
- How's that?

Clock Edges



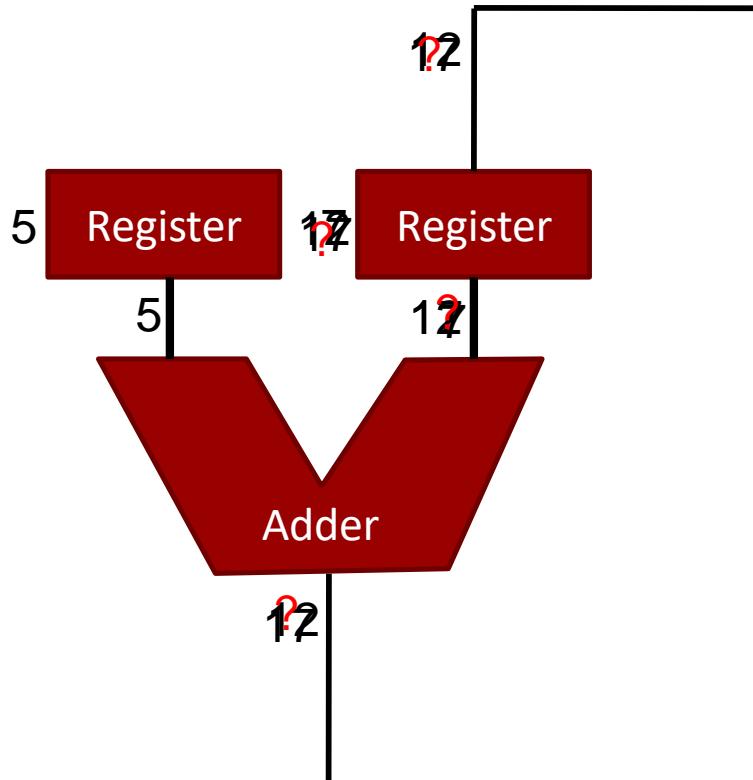
The Problem



And here we stay until the clock transitions

Clock signal **Low**
Registers output recorded value

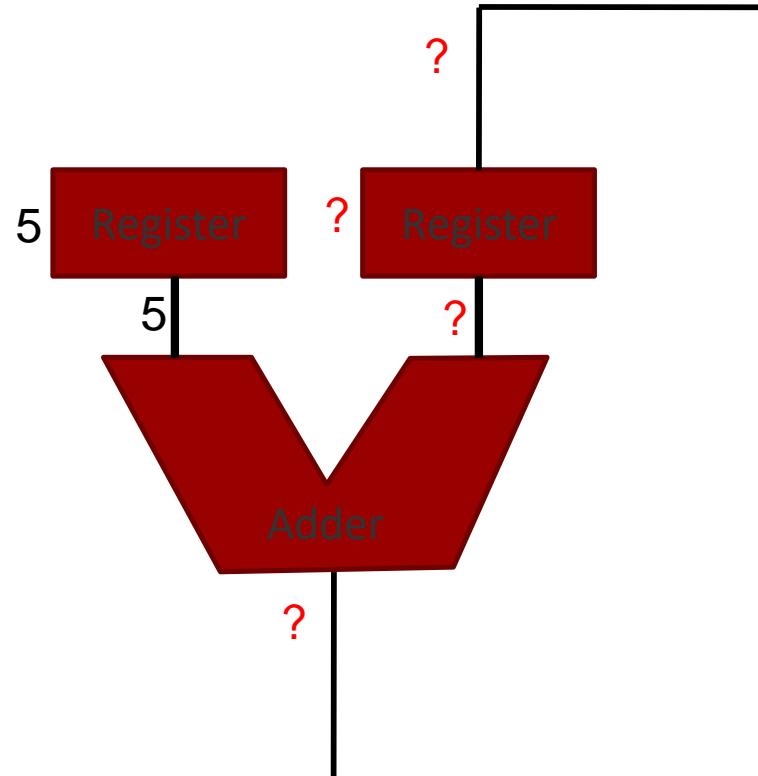
The Problem



And we stop when??

Clock signal **High**
Registers output recorded value

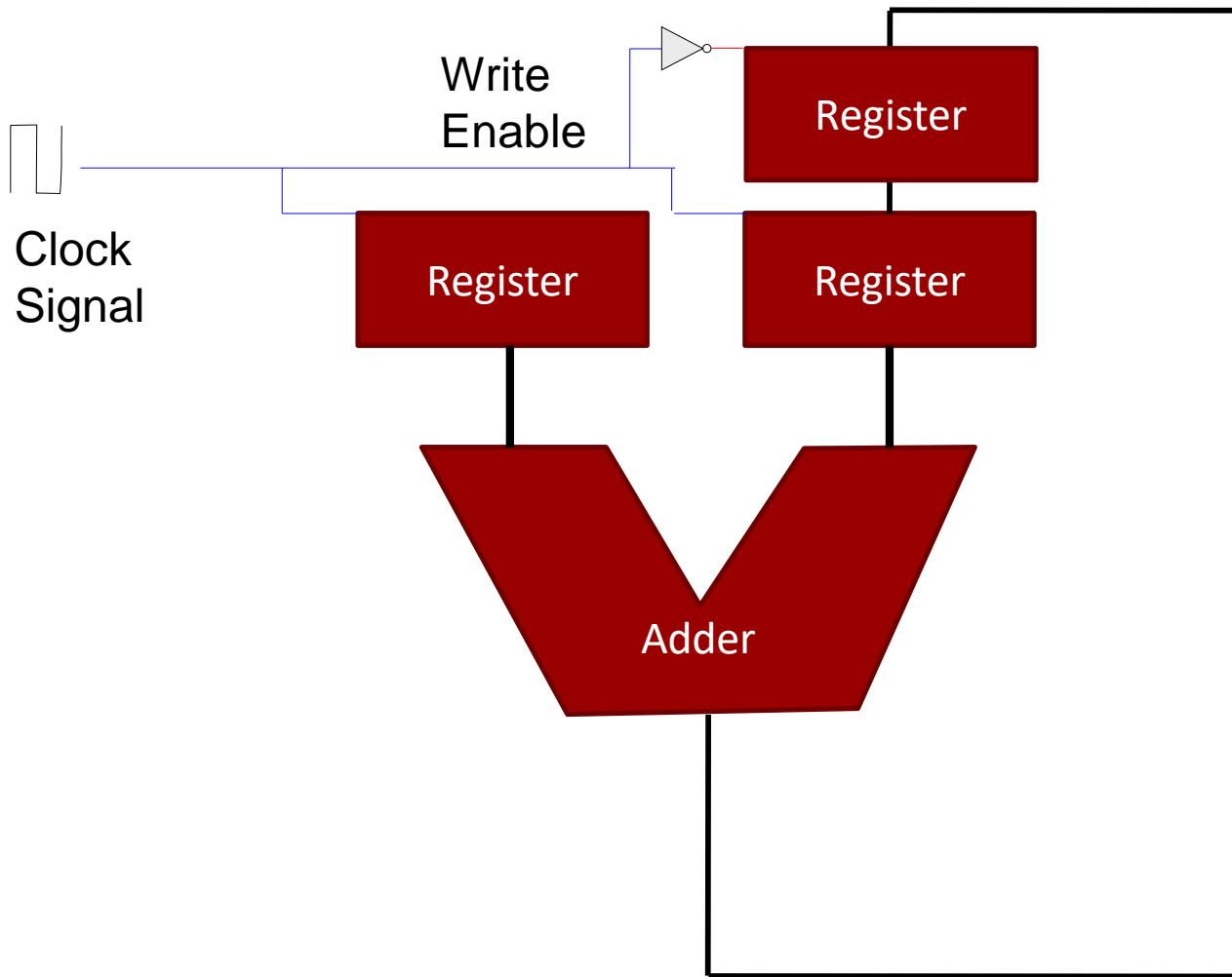
The Problem



And what value is in the register?

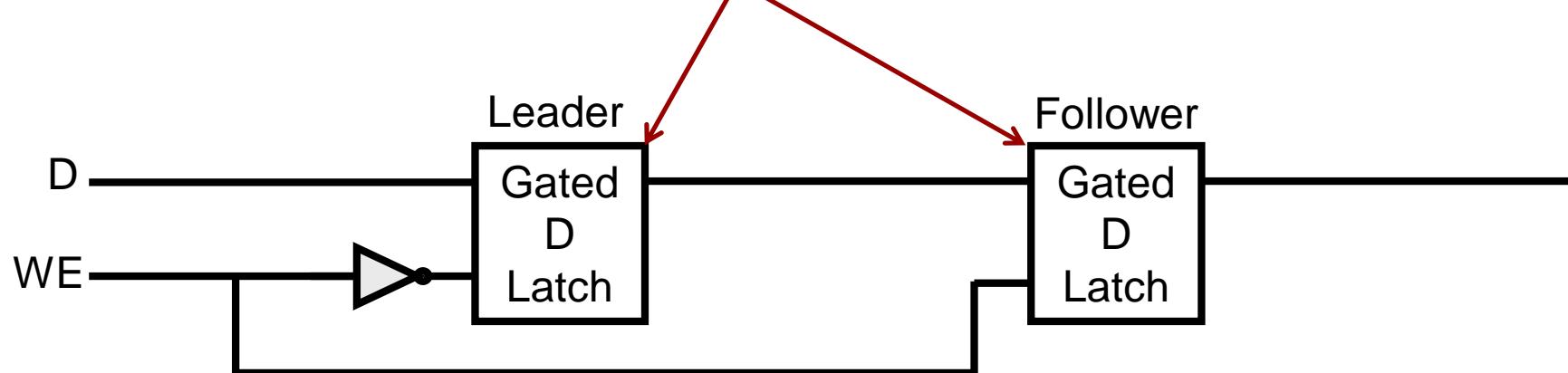
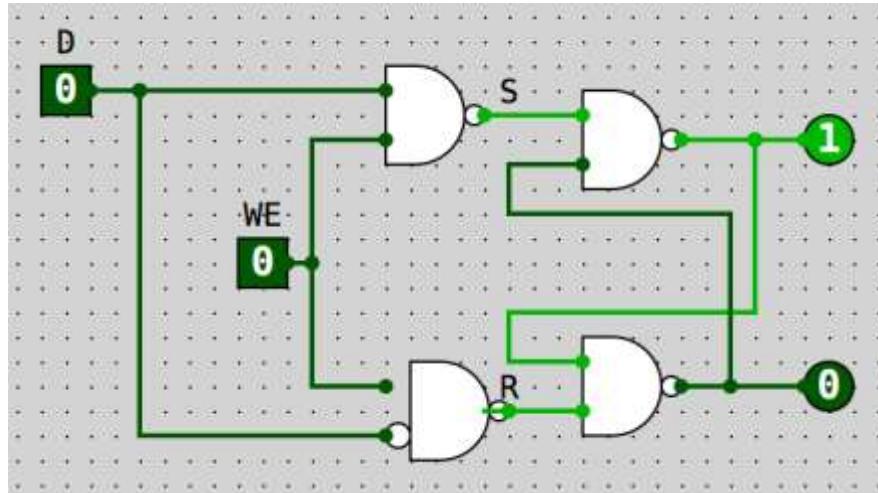
Clock signal **LOW**
Registers output recorded value

Possible Solution?



Leader-Follower Flip Flop

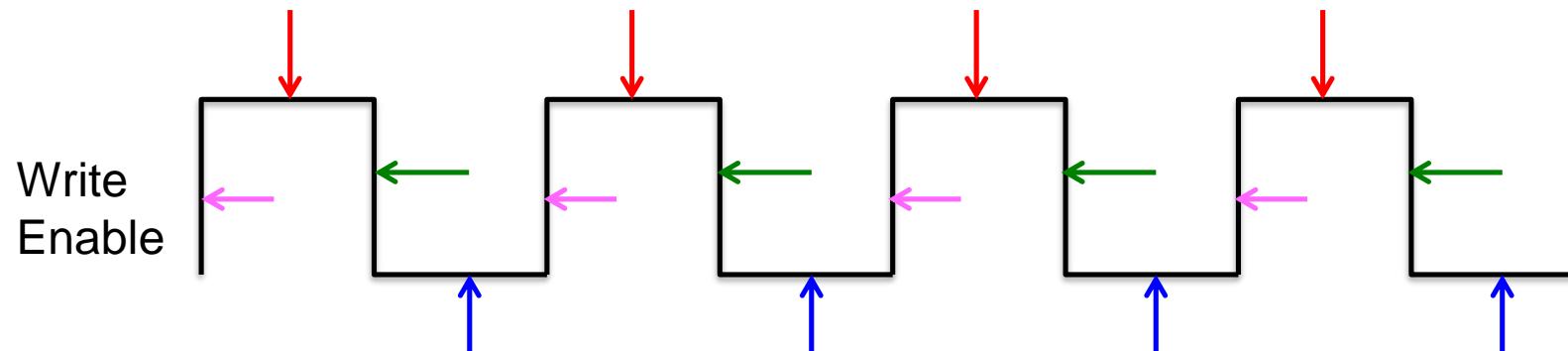
Remember our
Gated D Latch?



In the textbook this circuit is referred to as a Master/Slave flip flop

Clock Edges and Leader-Follower

Leader register records what is presented on its input.



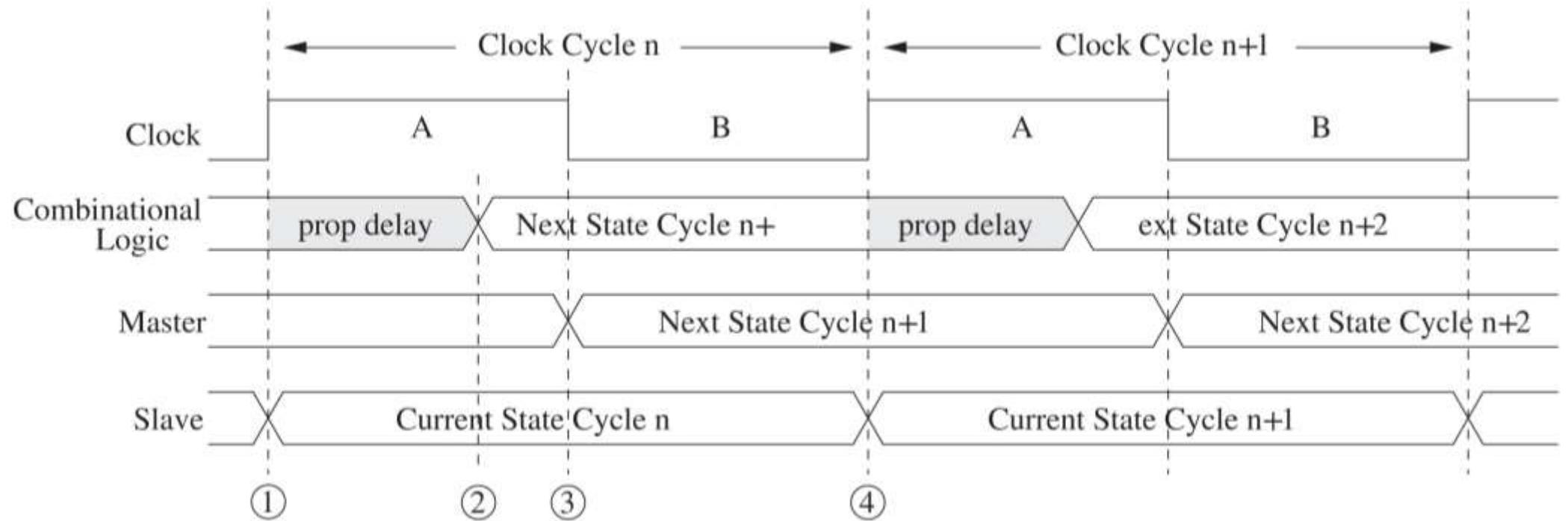
Follower register records what is presented on its input.

Follower register outputs its inputs.
Leader register outputs what was recorded when WE rose to 1.

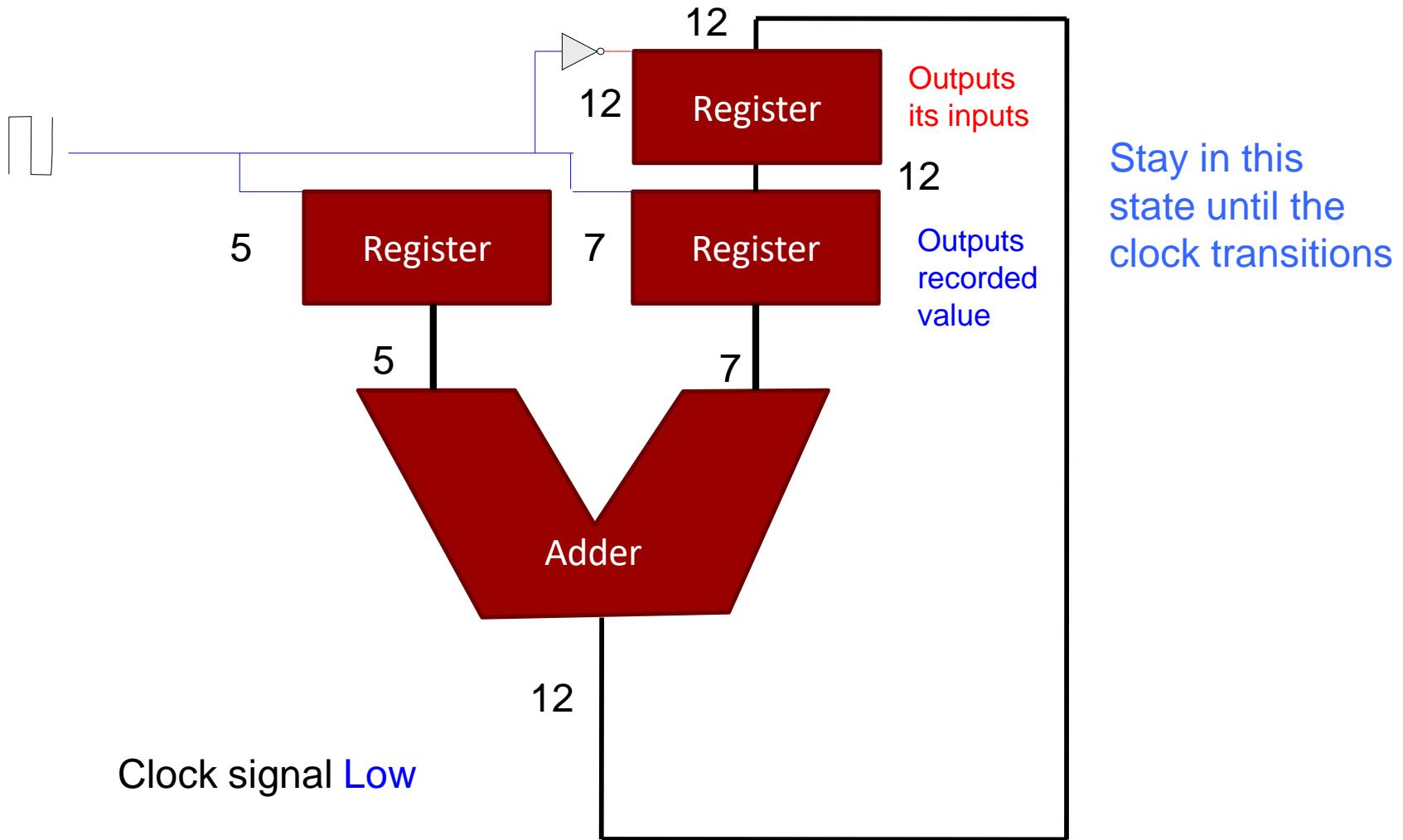
Follower register outputs what was recorded when WE fell to 0.
Leader register outputs its inputs.

Now we just connect Write Enable to the clock....

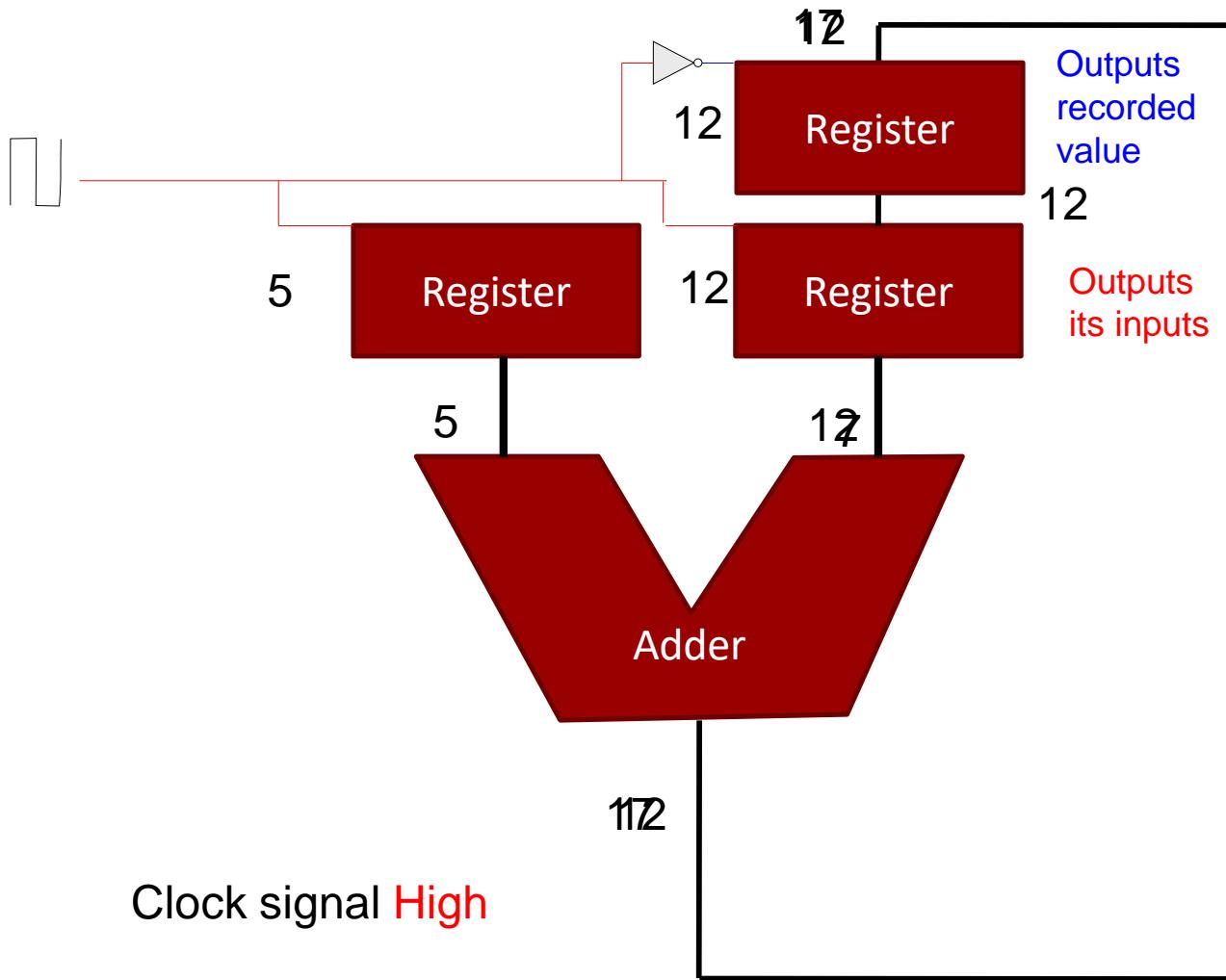
Another way to look at it



Possible Solution?

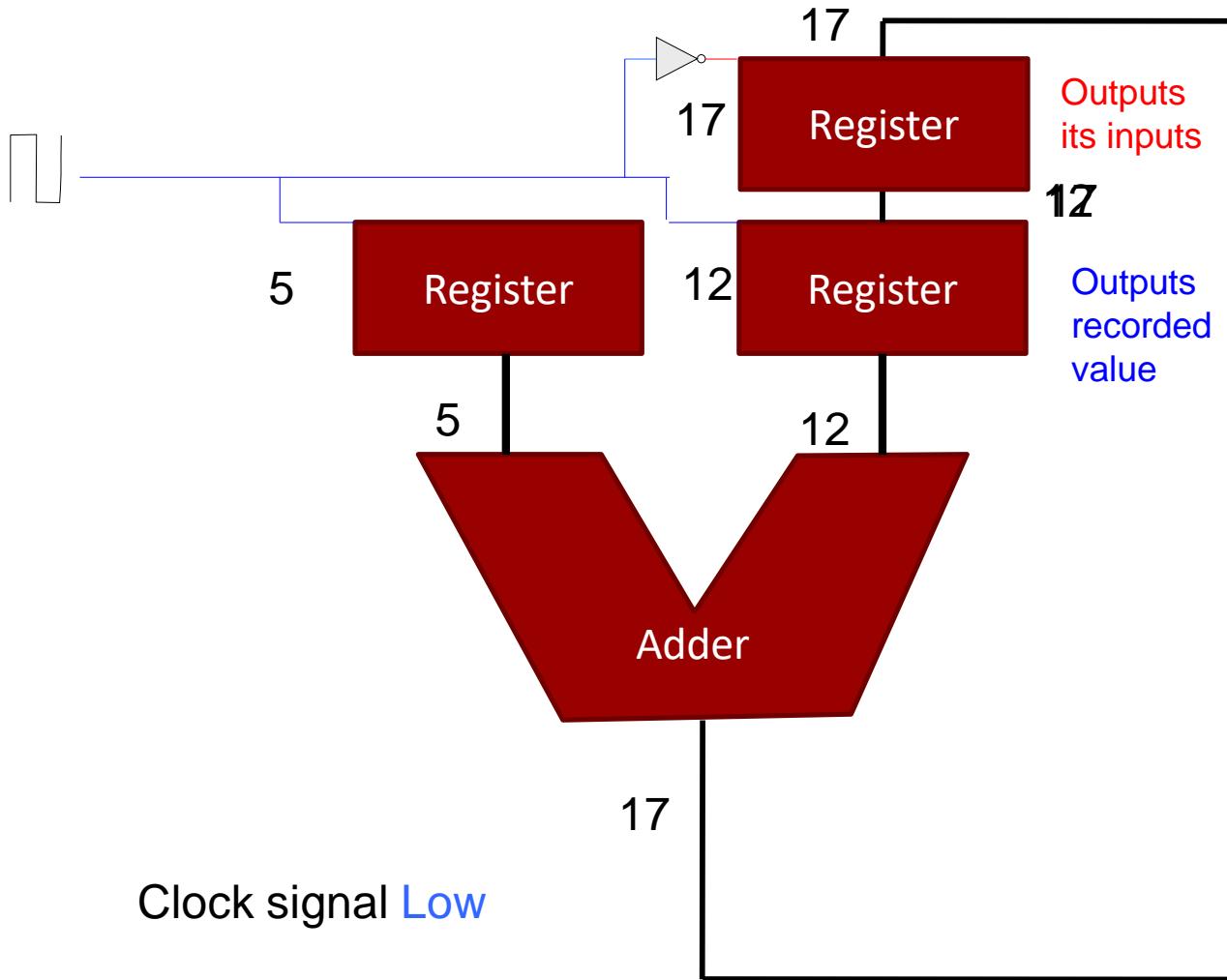


Possible Solution?



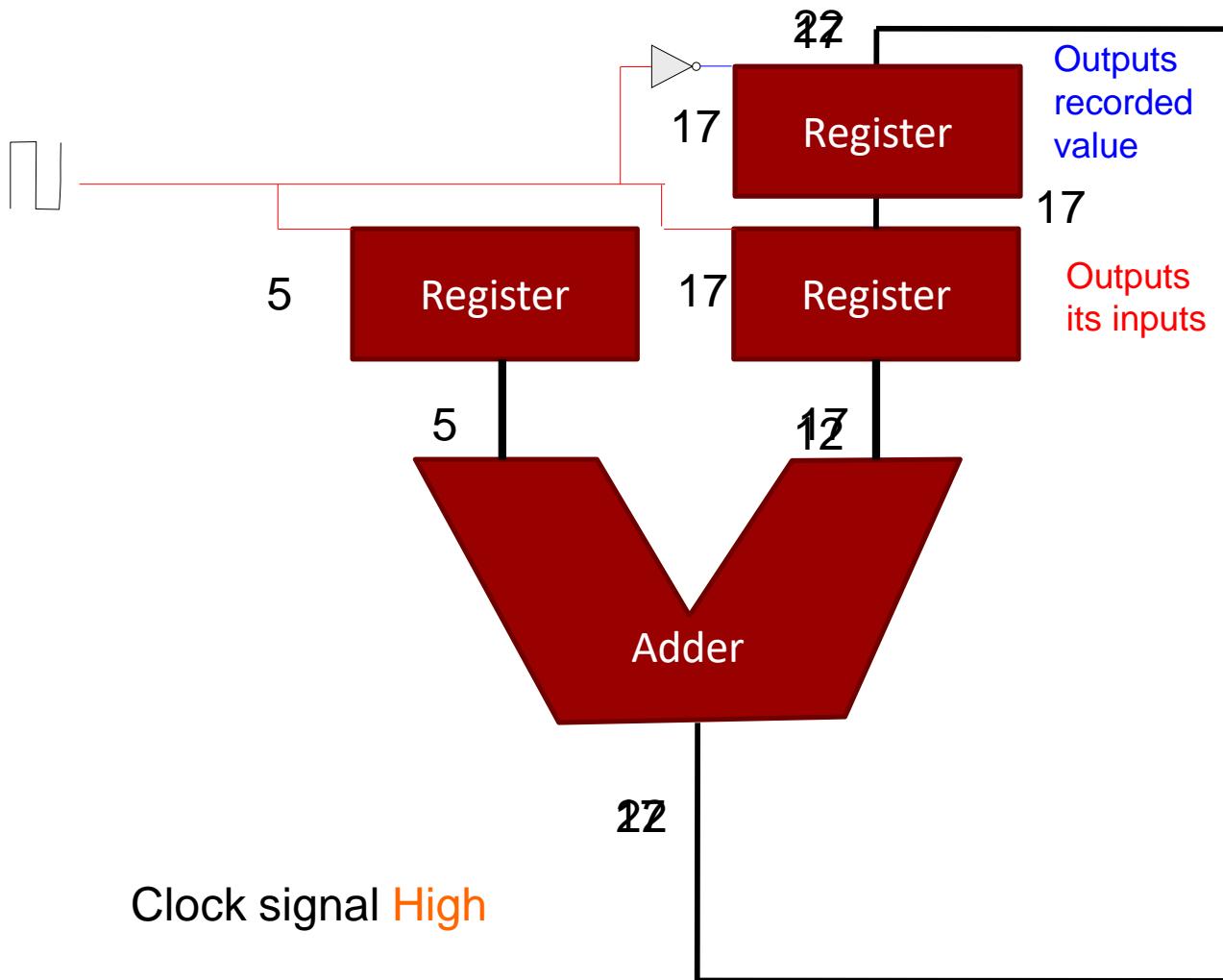
Stay in this state until the clock transitions

Possible Solution?



Stay in this state until the clock transitions

Possible Solution?



Stay in this state until the clock transitions

Terminology

Patt

Gated D Latch

Leader-Follower Flip-Flop

Master-Slave Flip-flop

Circuitsim

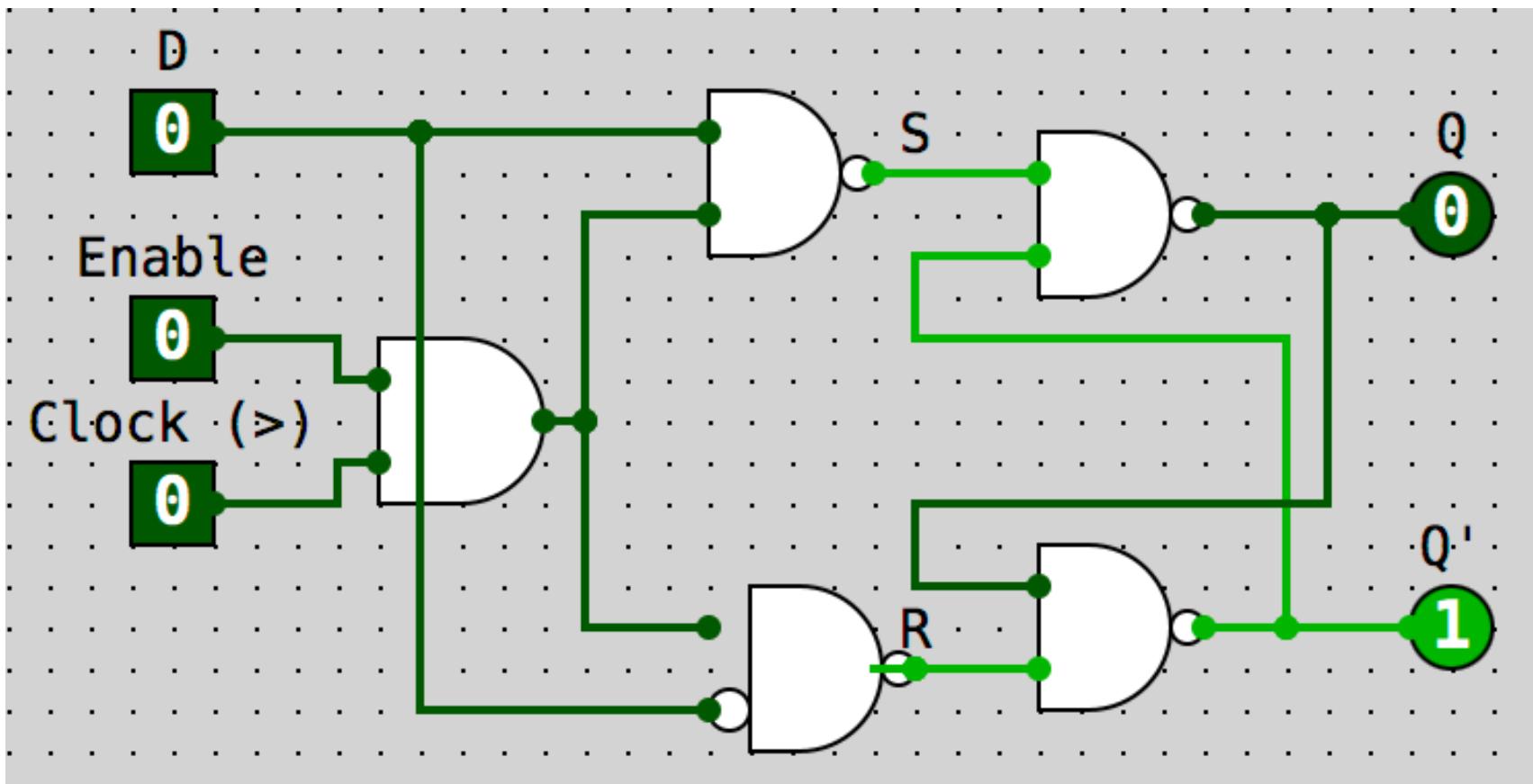
Build it yourself!

D Flip-Flop

-or-

Register

How Does Enable Work?



Question

Level-triggered sequential circuits can yield unexpected results if

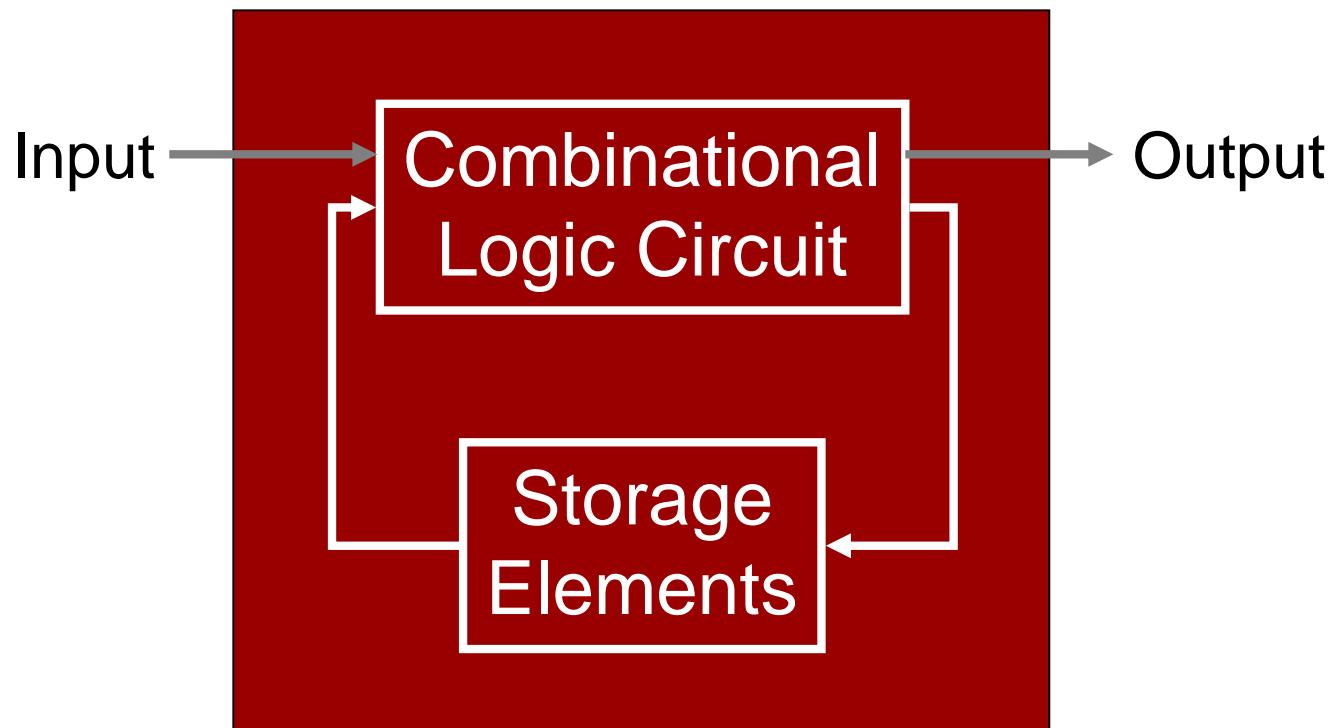
- A. The device is presented with different inputs in subsequent clock cycles
- B. The input of the device is derived from manually operated switches
- C. The clock signal to the device is stopped
- D. The output of the device is used in computing the input to the same device



Now For Finite State Machines

- Which is what we build with Sequential Logic...

Sequential Logic Circuit

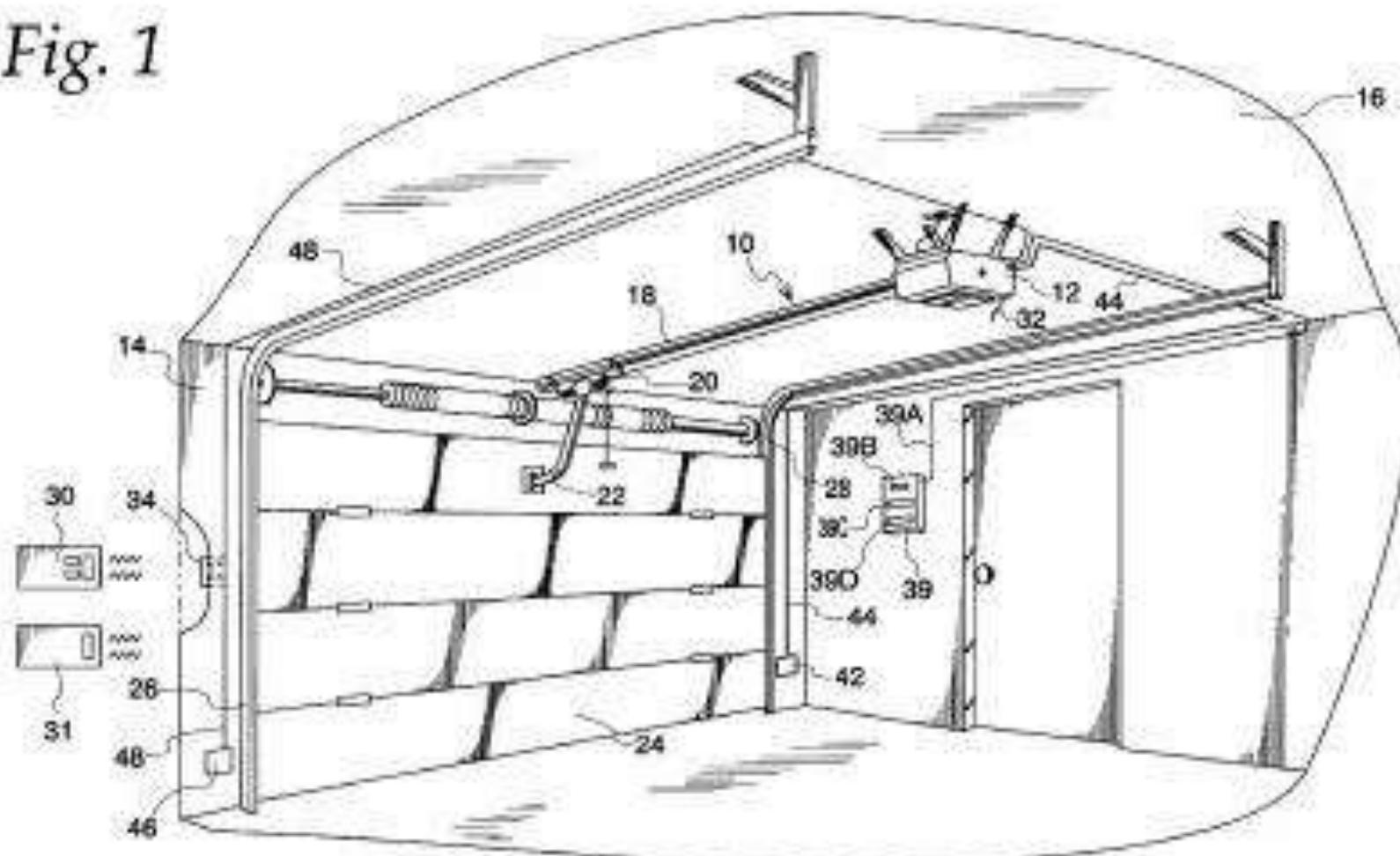


Concept of State

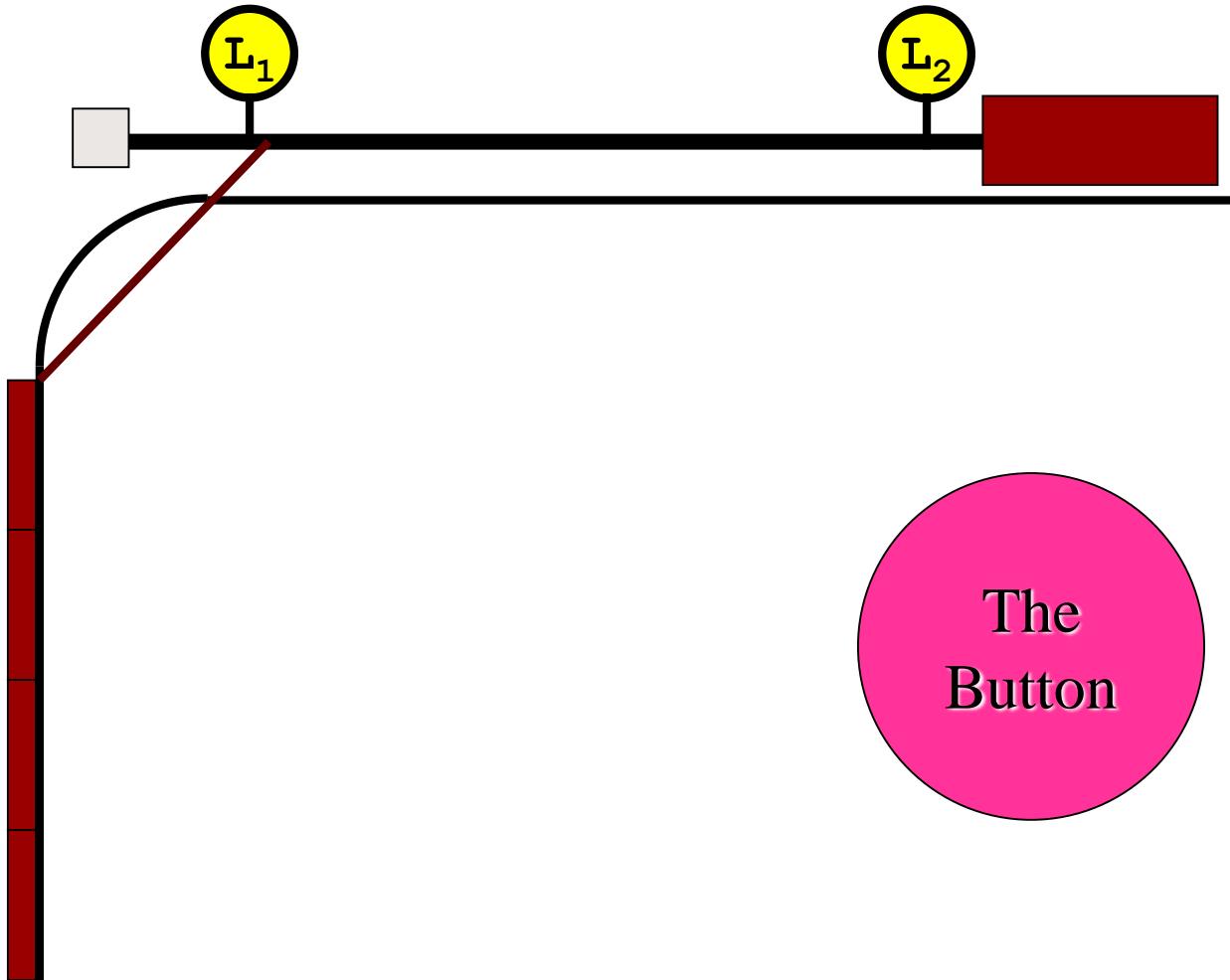
- Vending Machines
- Adding Machines
- Traffic Signal Controls
- Parking Lot Controls
- Combination Locks
- Computers
- Garage Door Openers

A Garage Door Opener

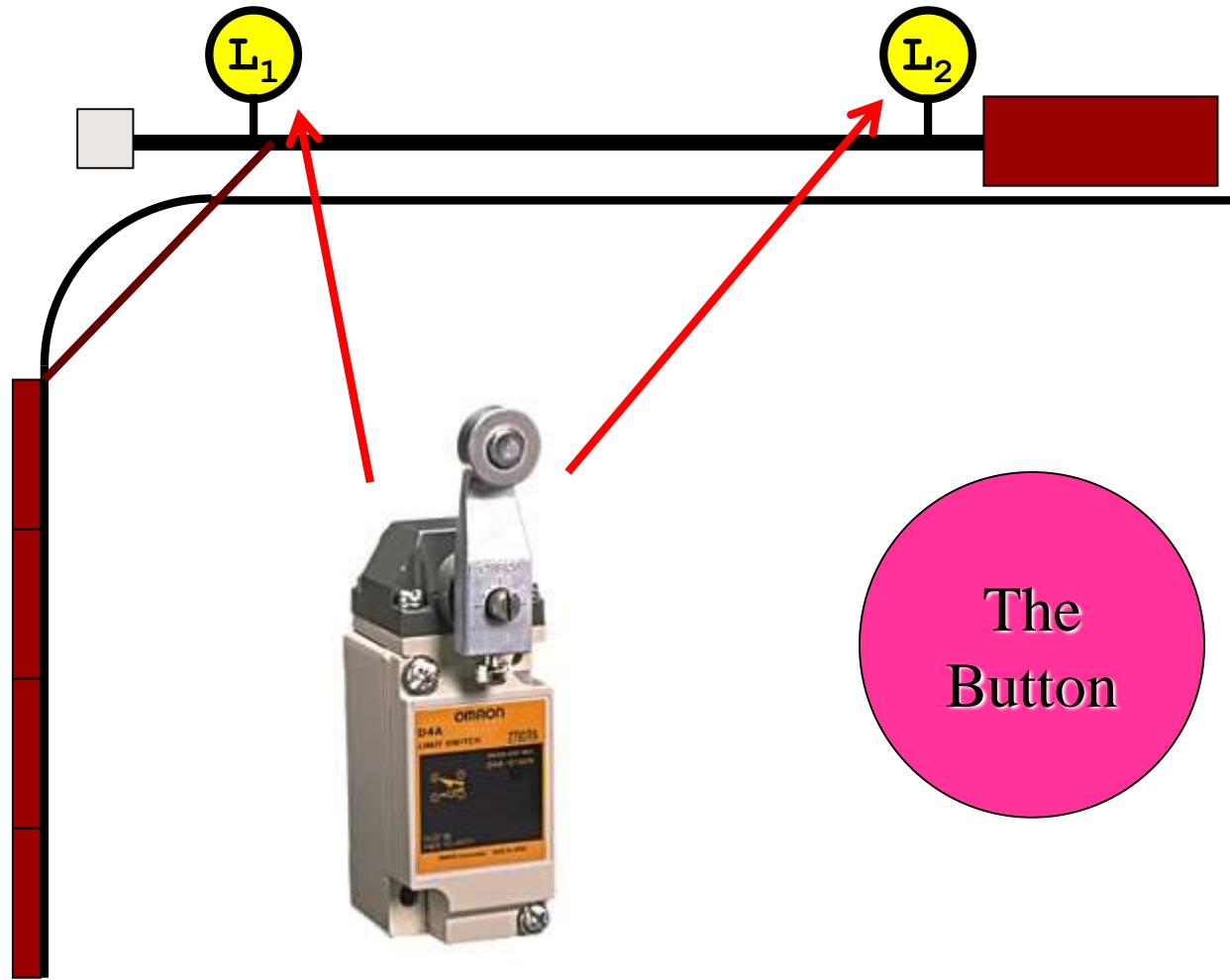
Fig. 1



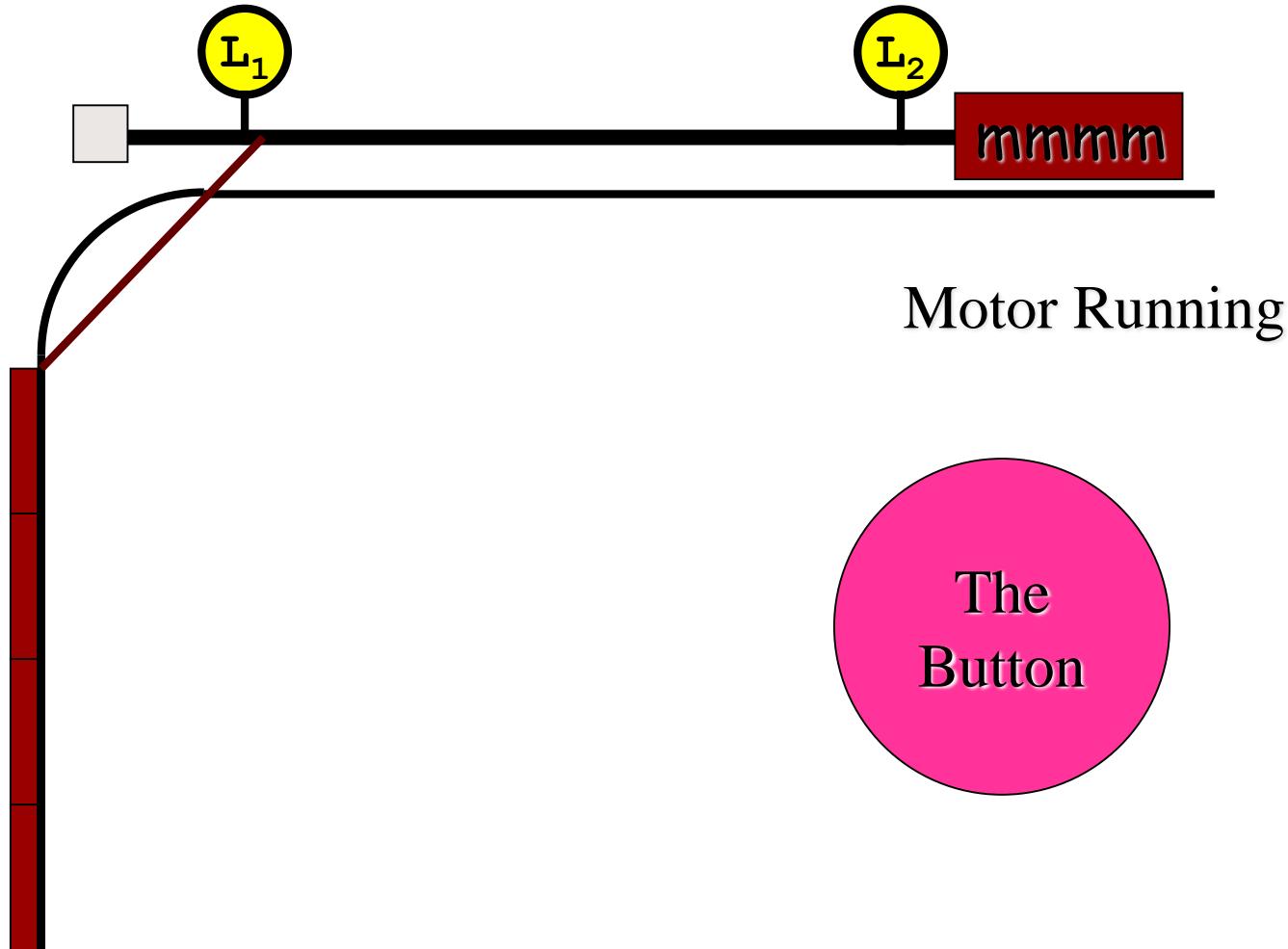
My Garage Door



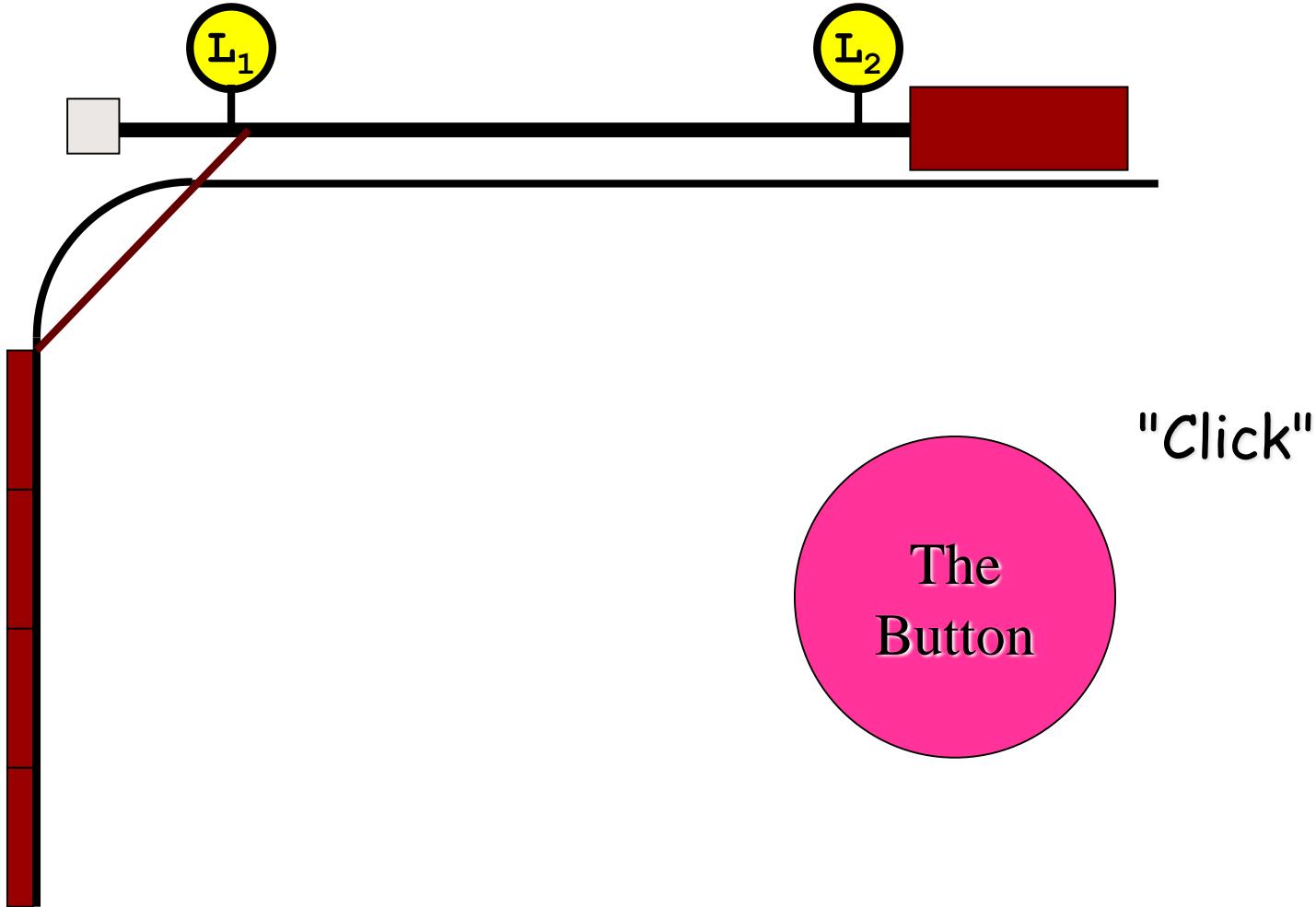
My Garage Door



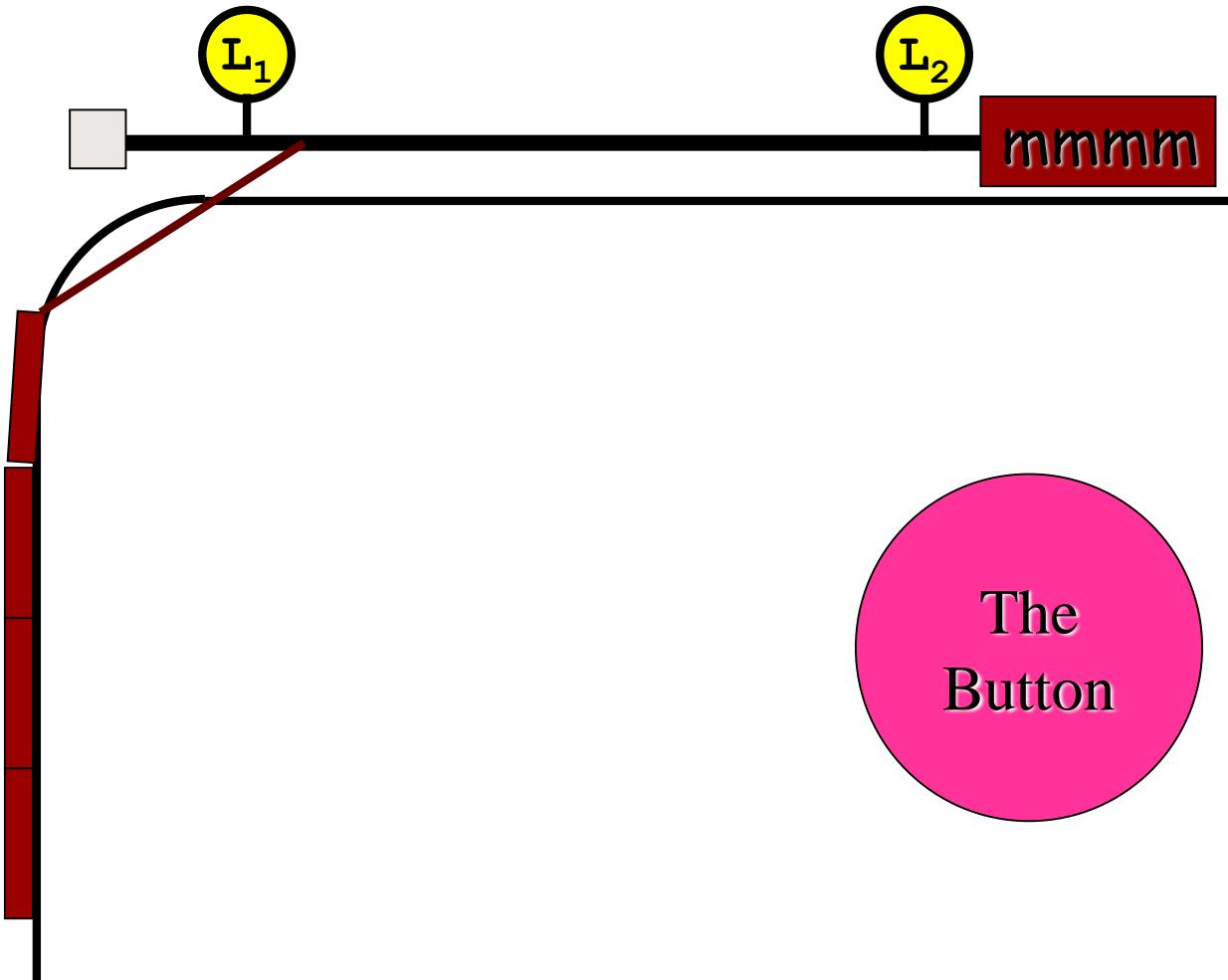
My Garage Door



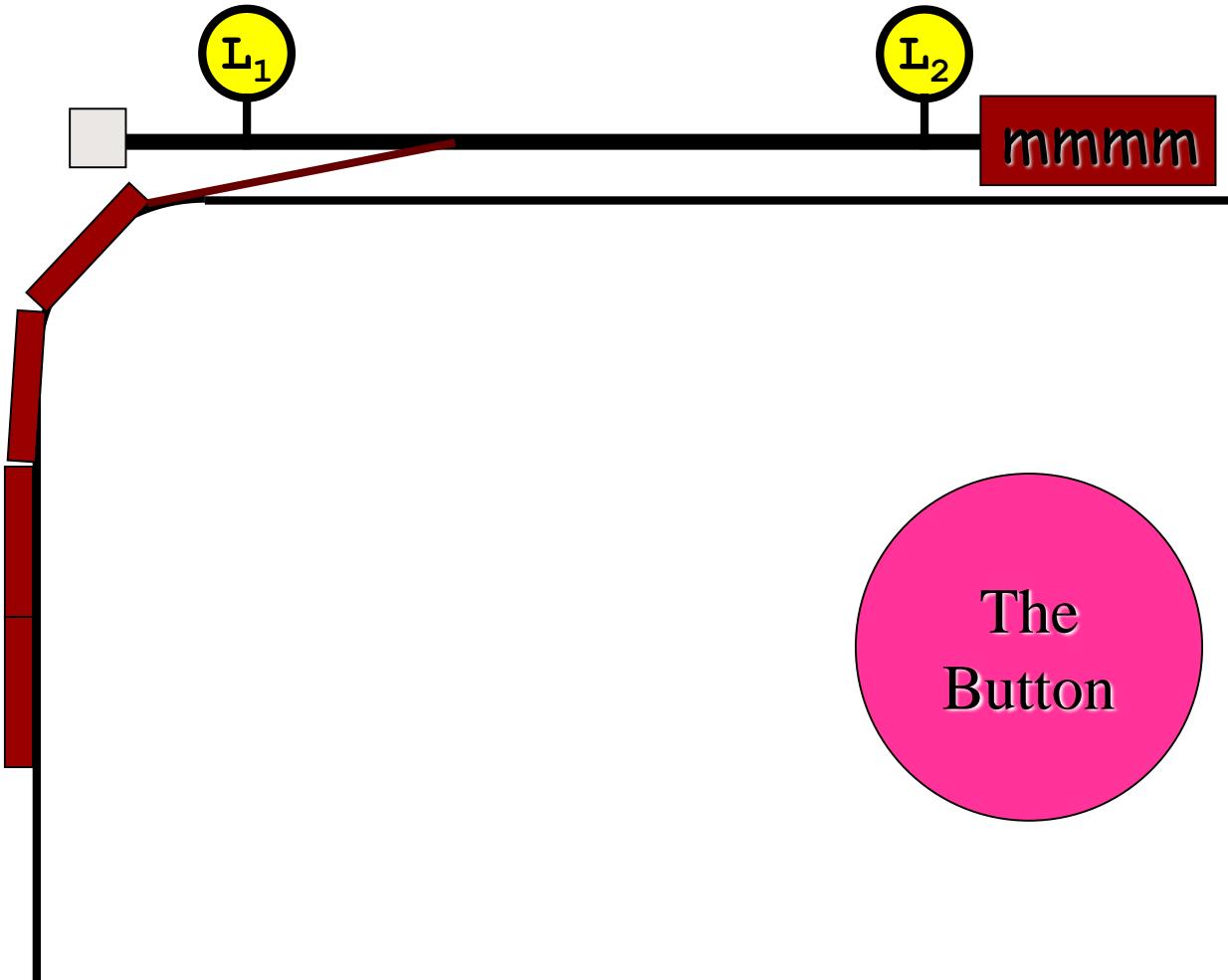
My Garage Door



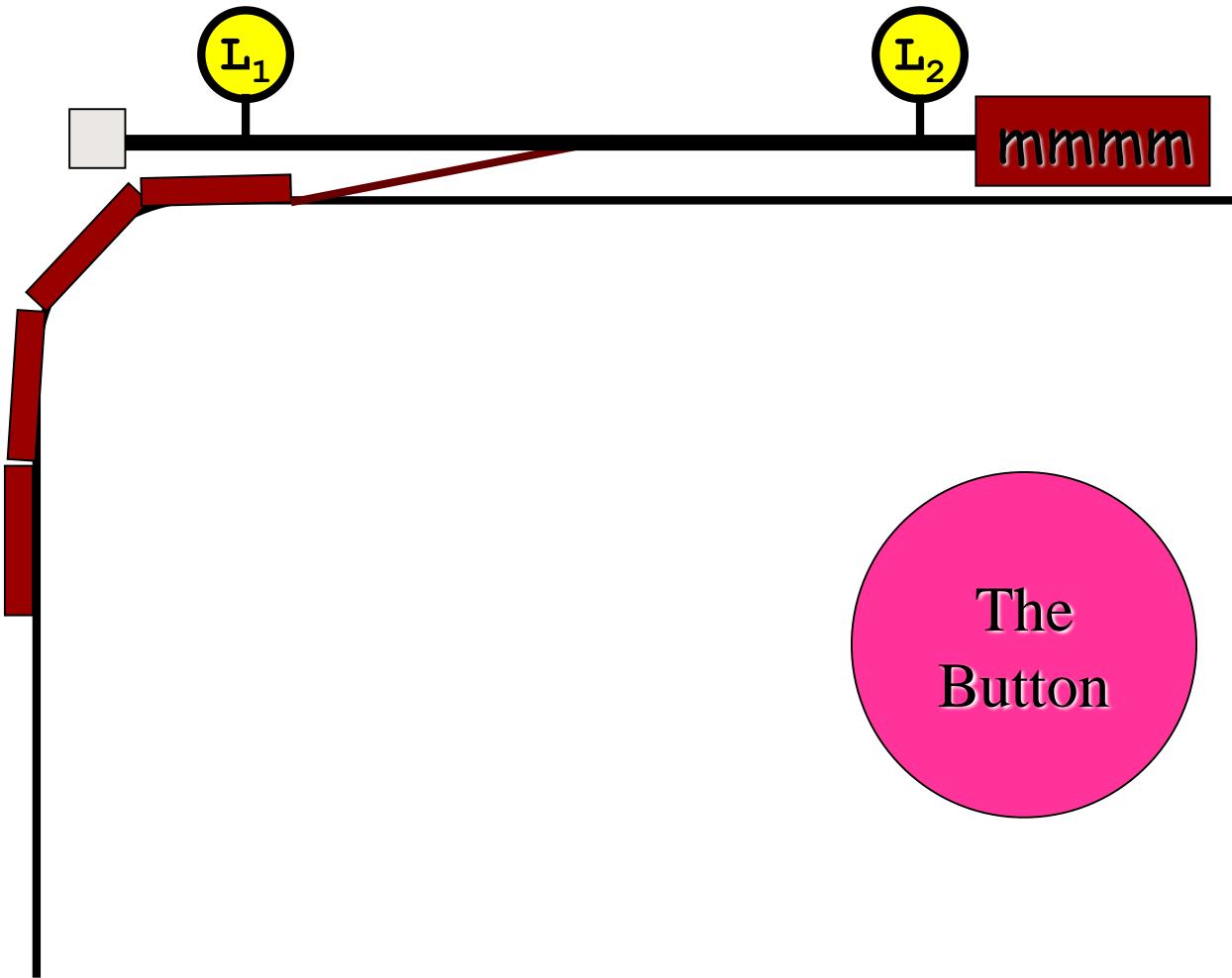
My Garage Door



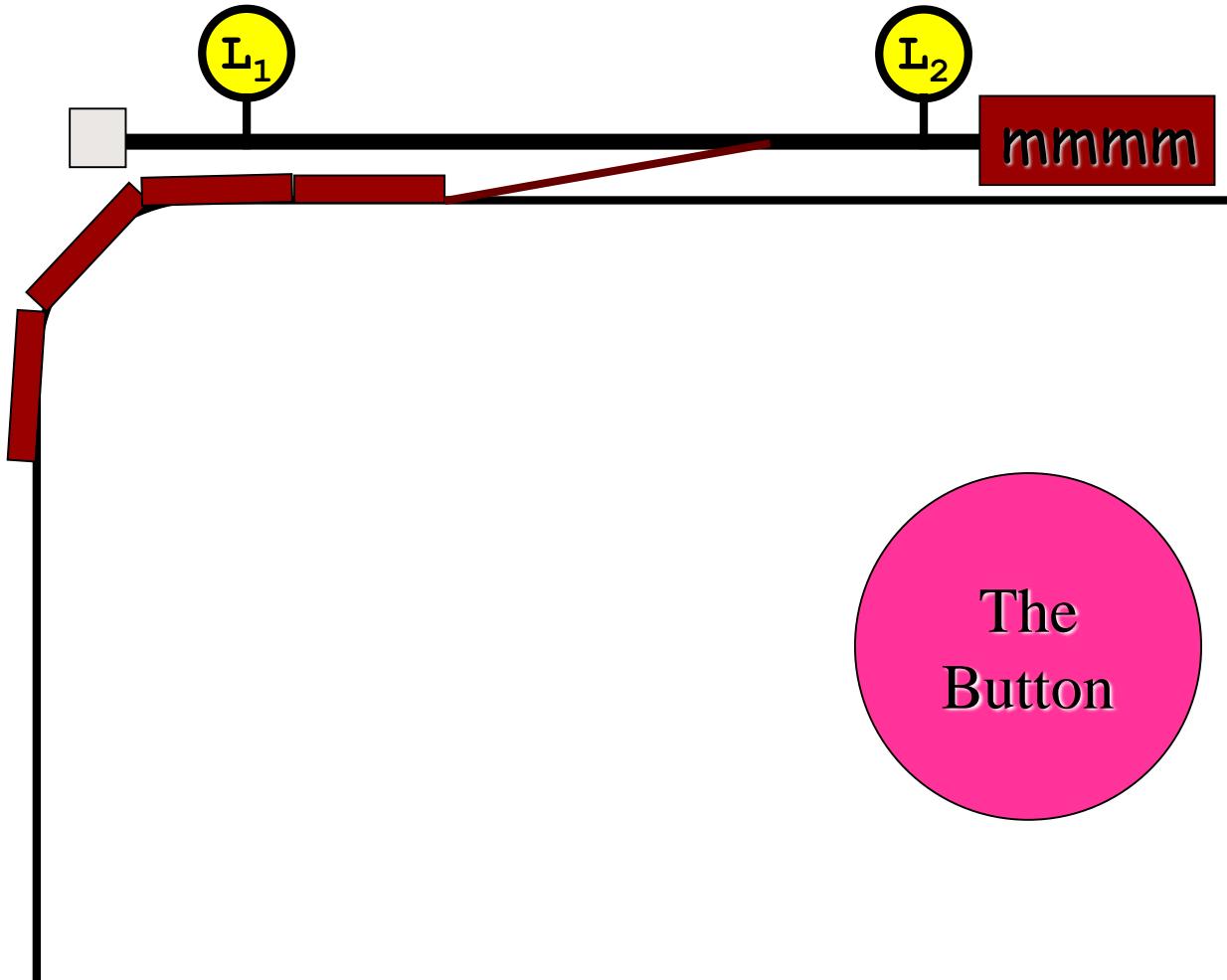
My Garage Door



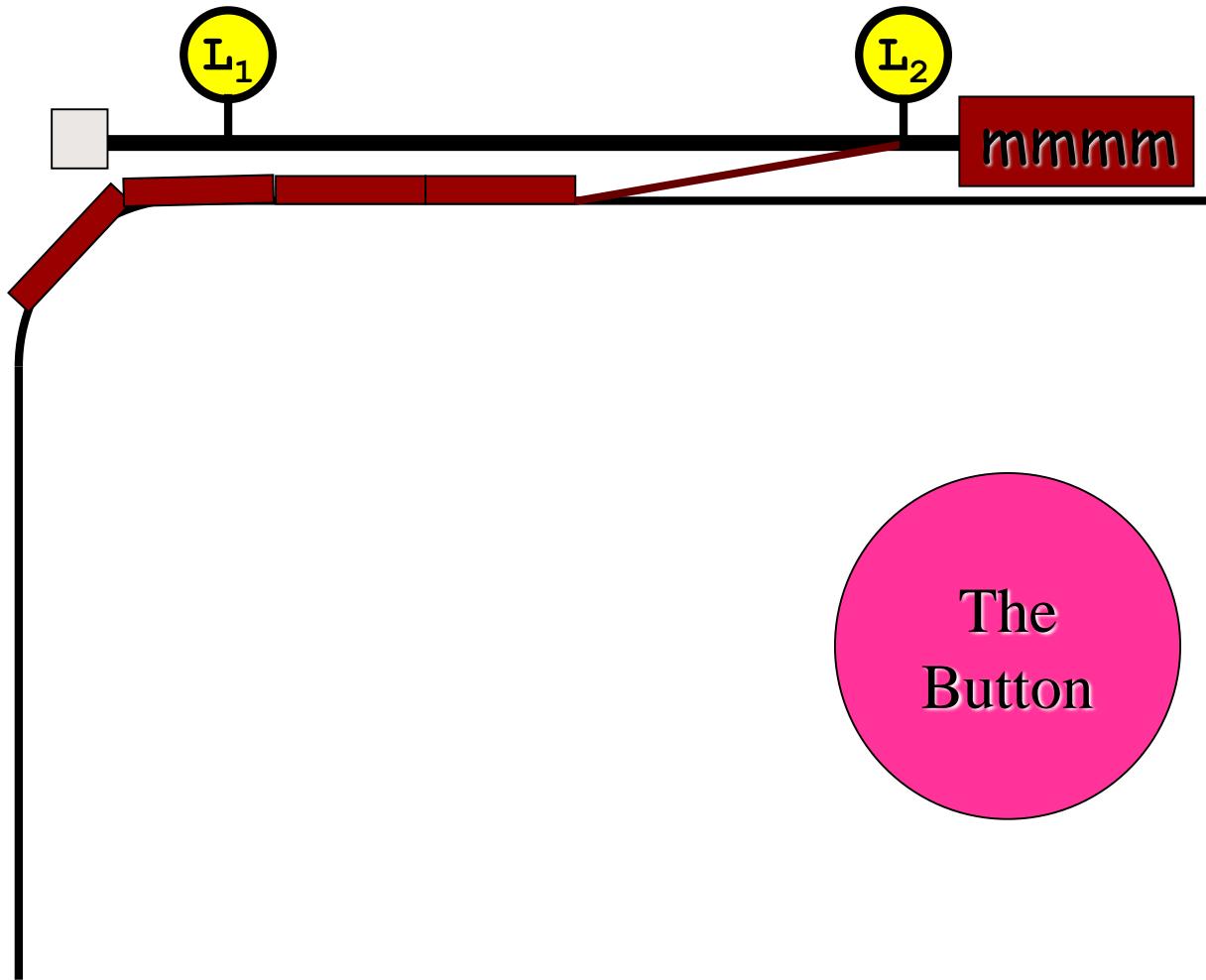
My Garage Door



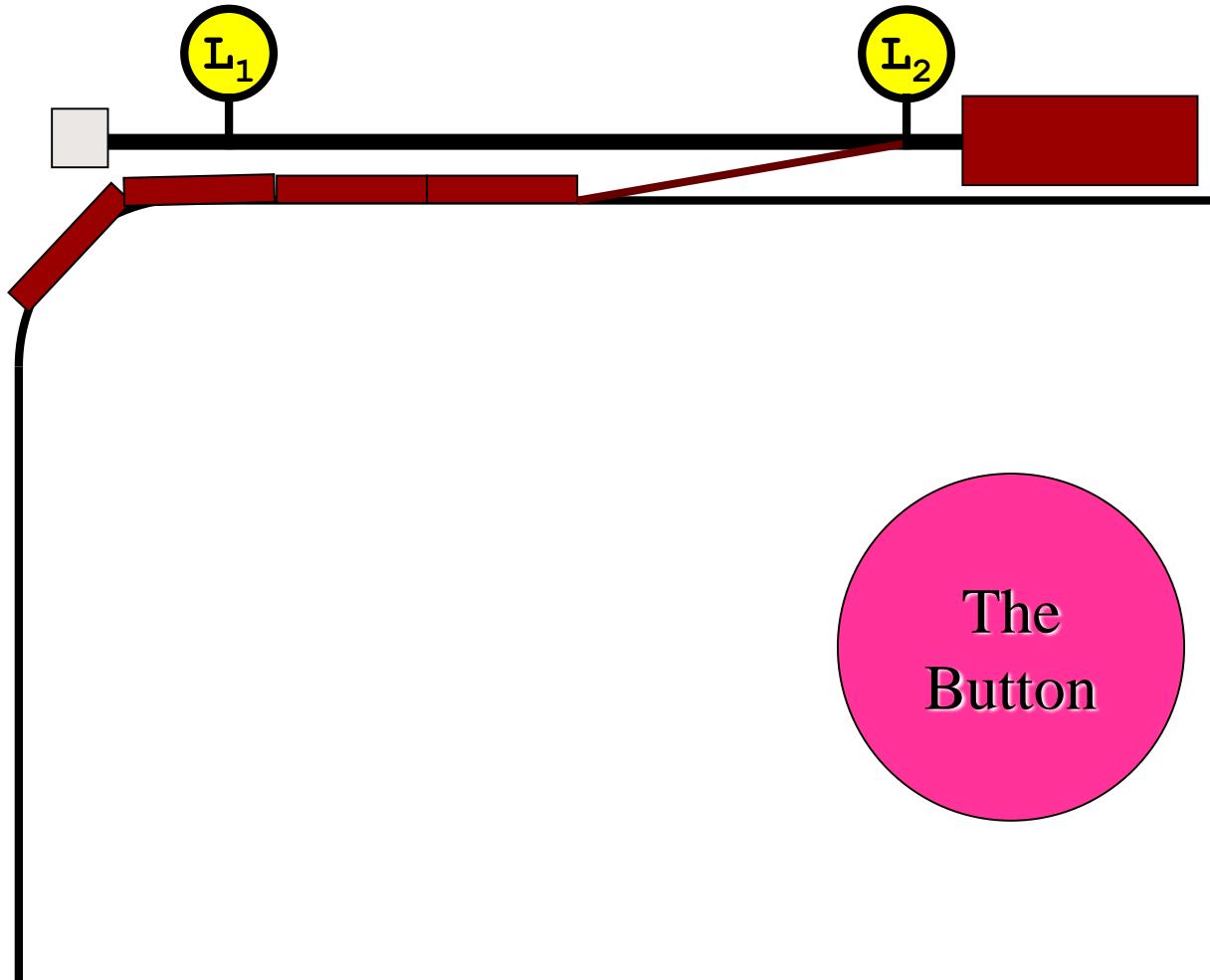
My Garage Door



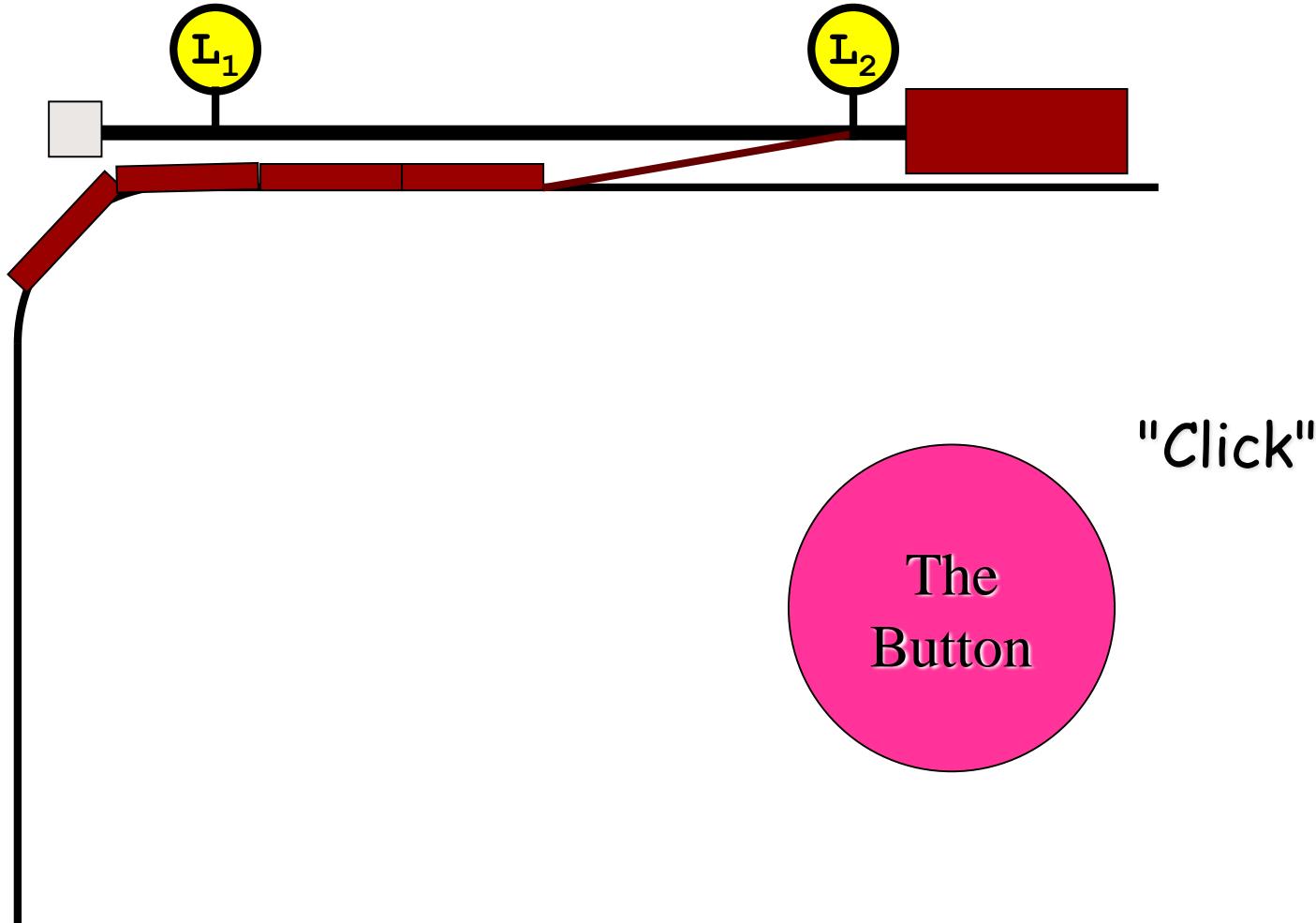
My Garage Door



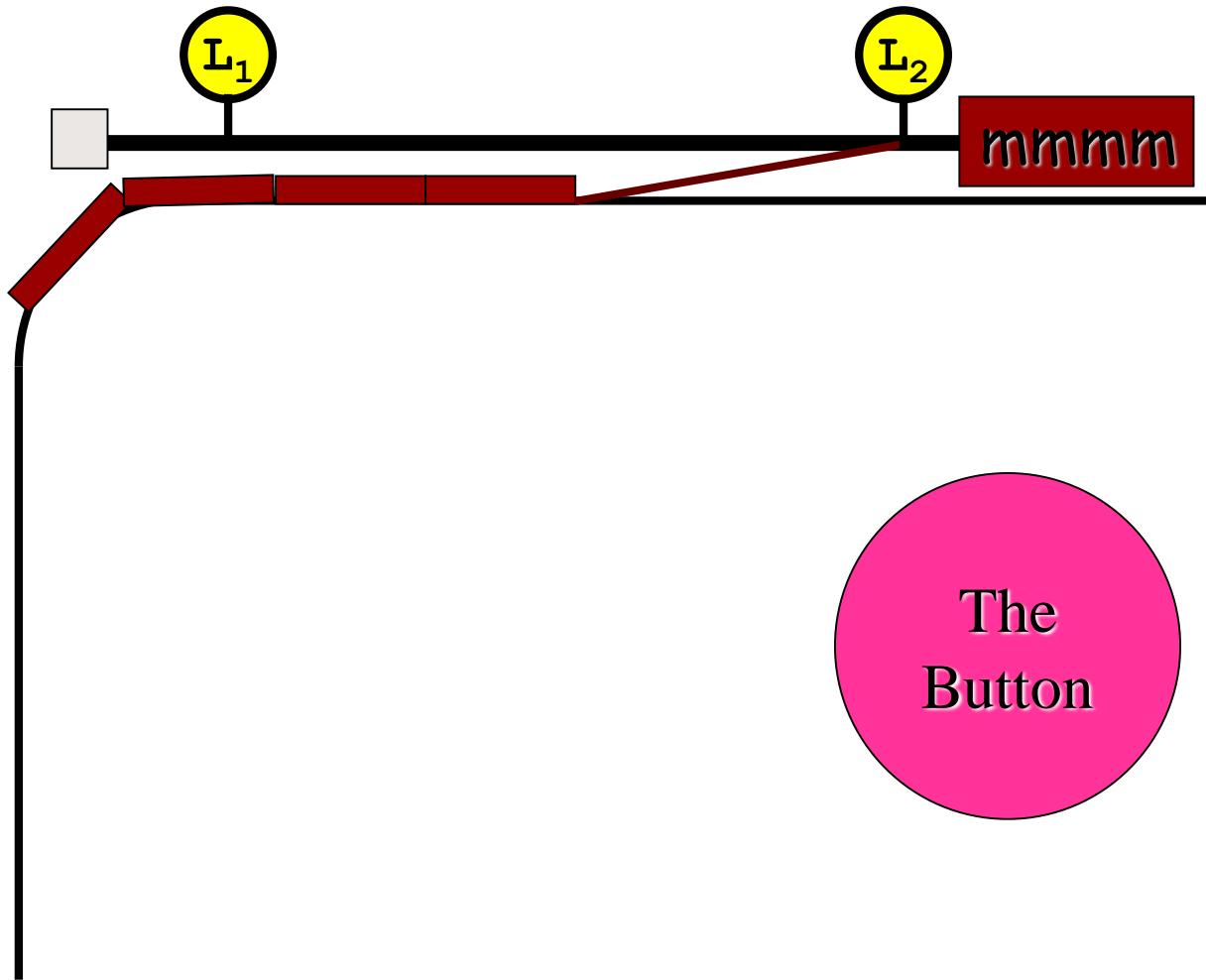
My Garage Door



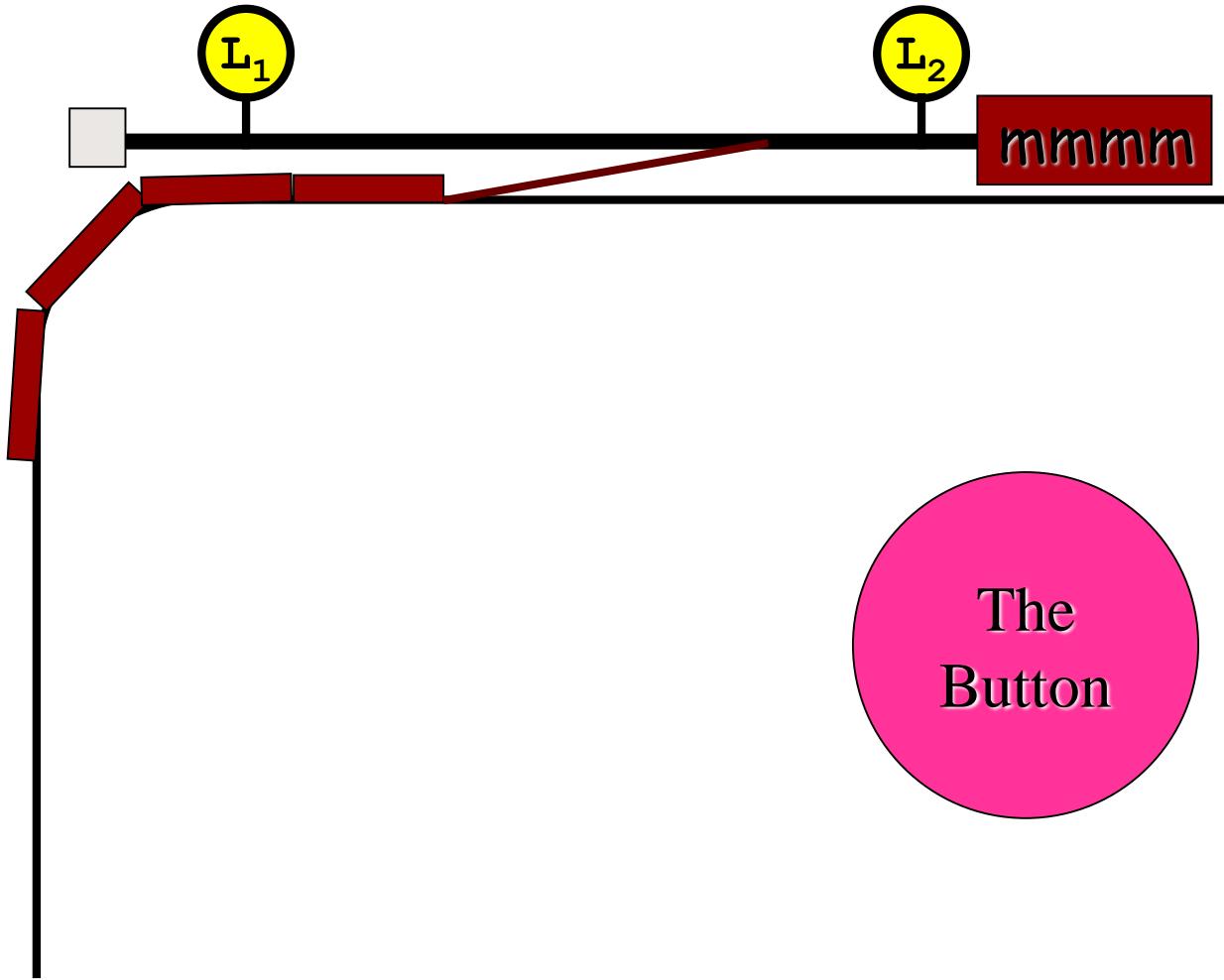
My Garage Door



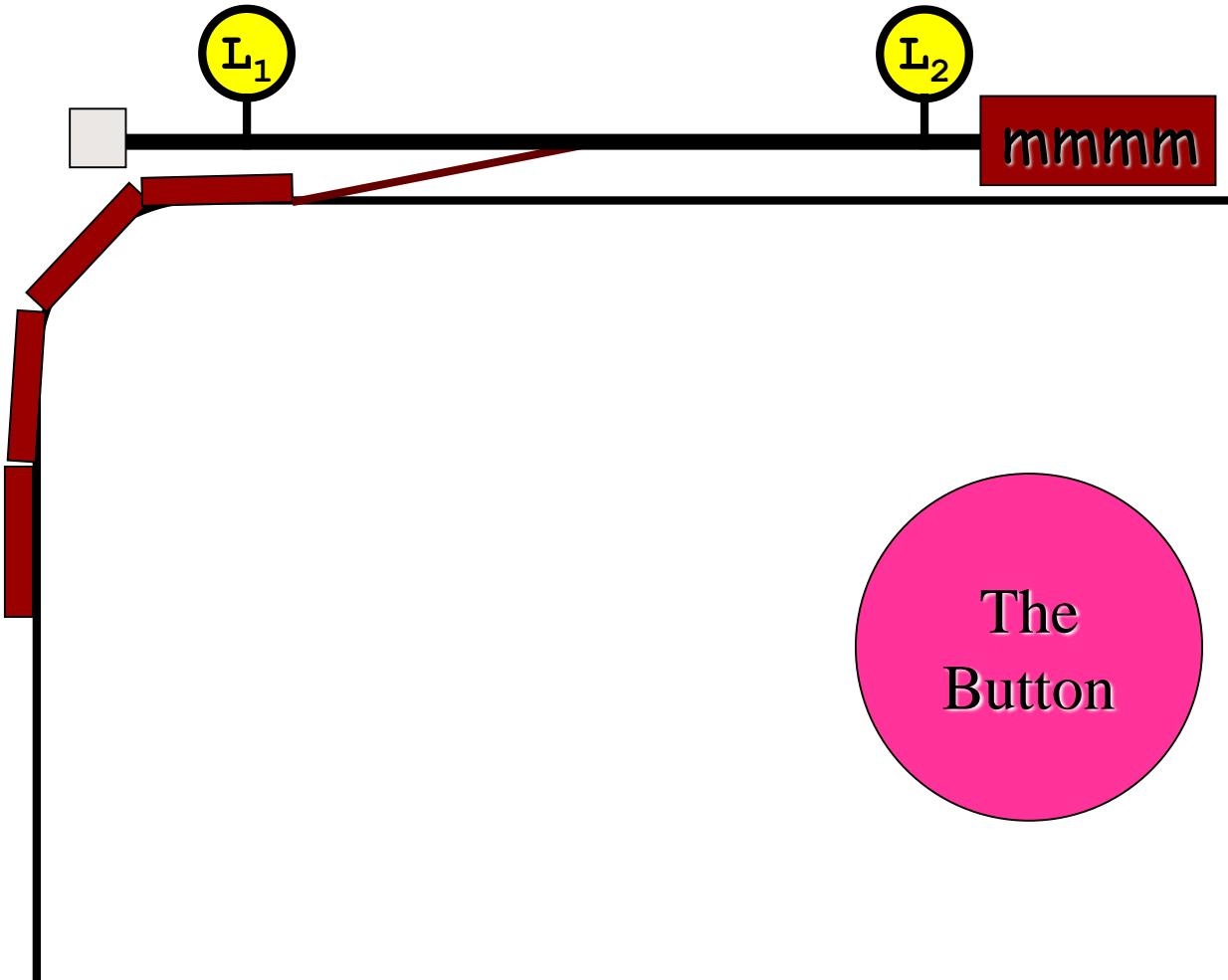
My Garage Door



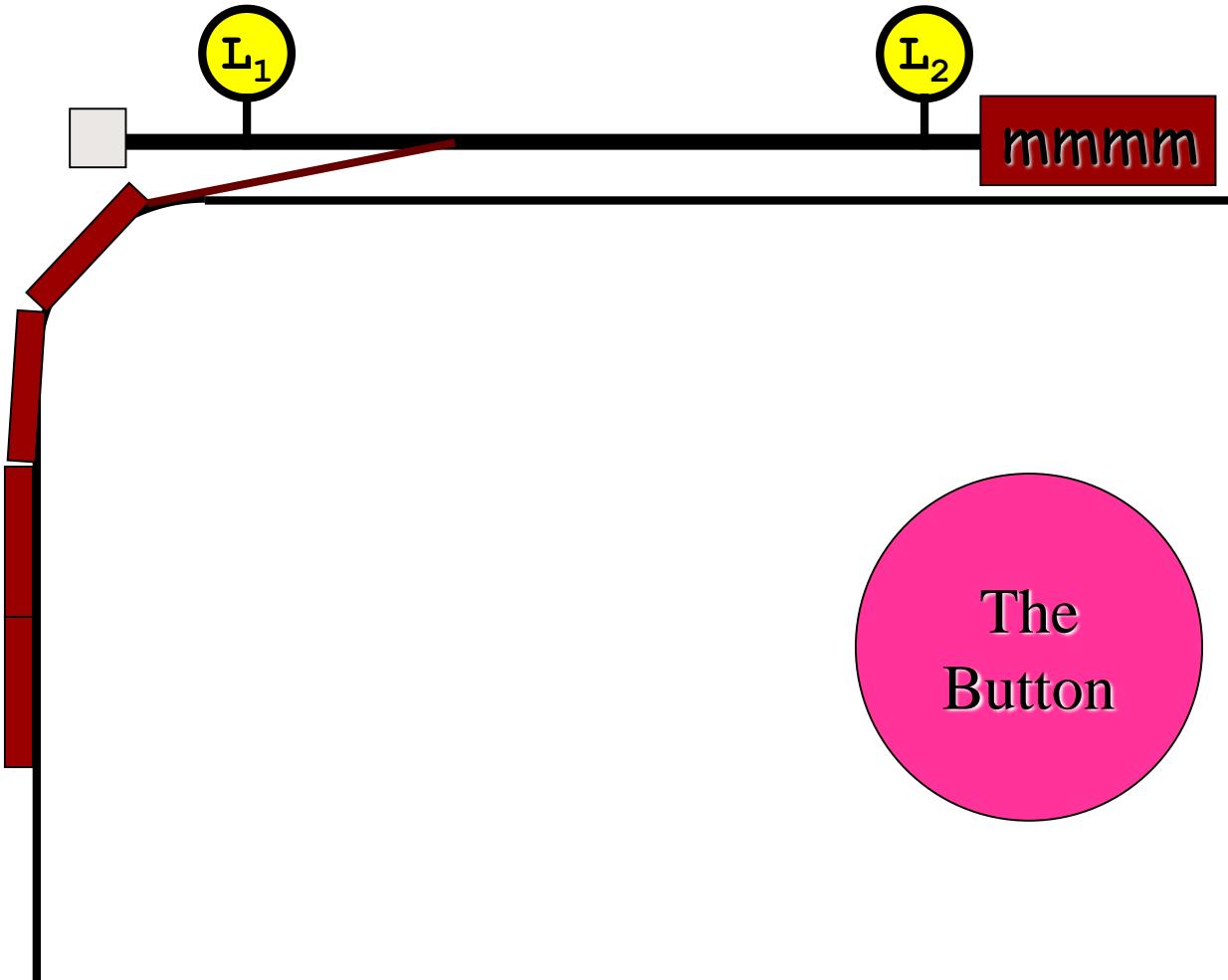
My Garage Door



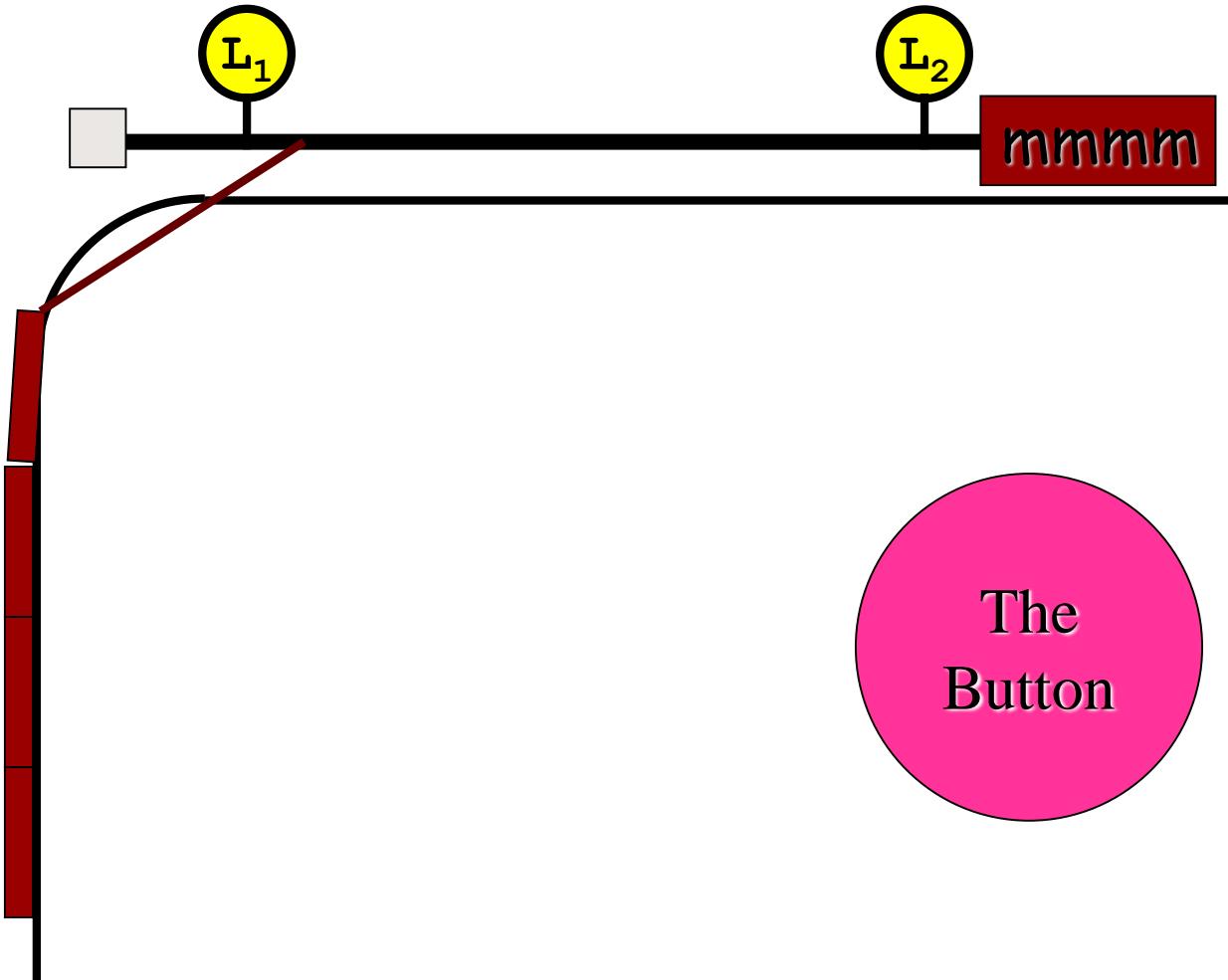
My Garage Door



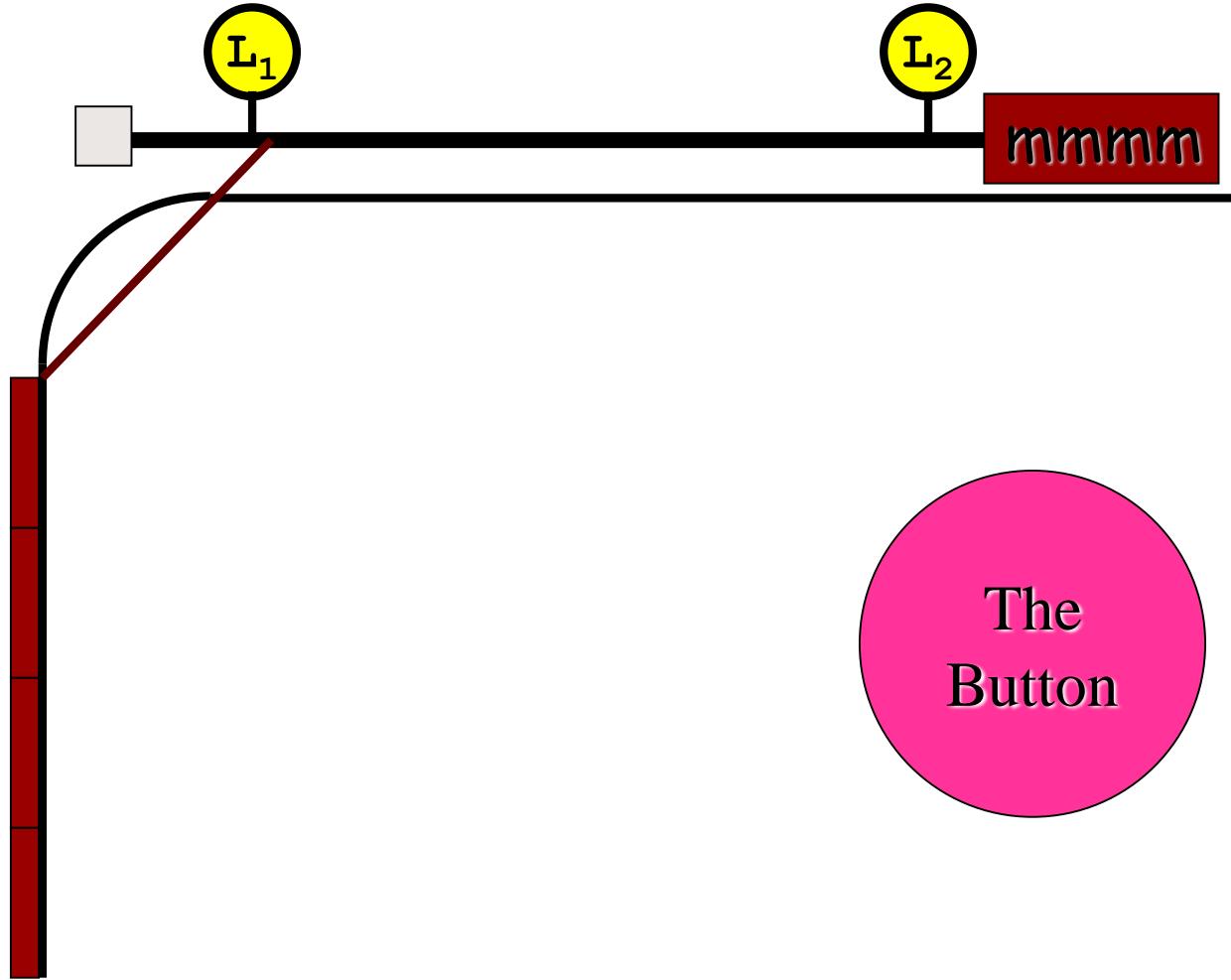
My Garage Door



My Garage Door

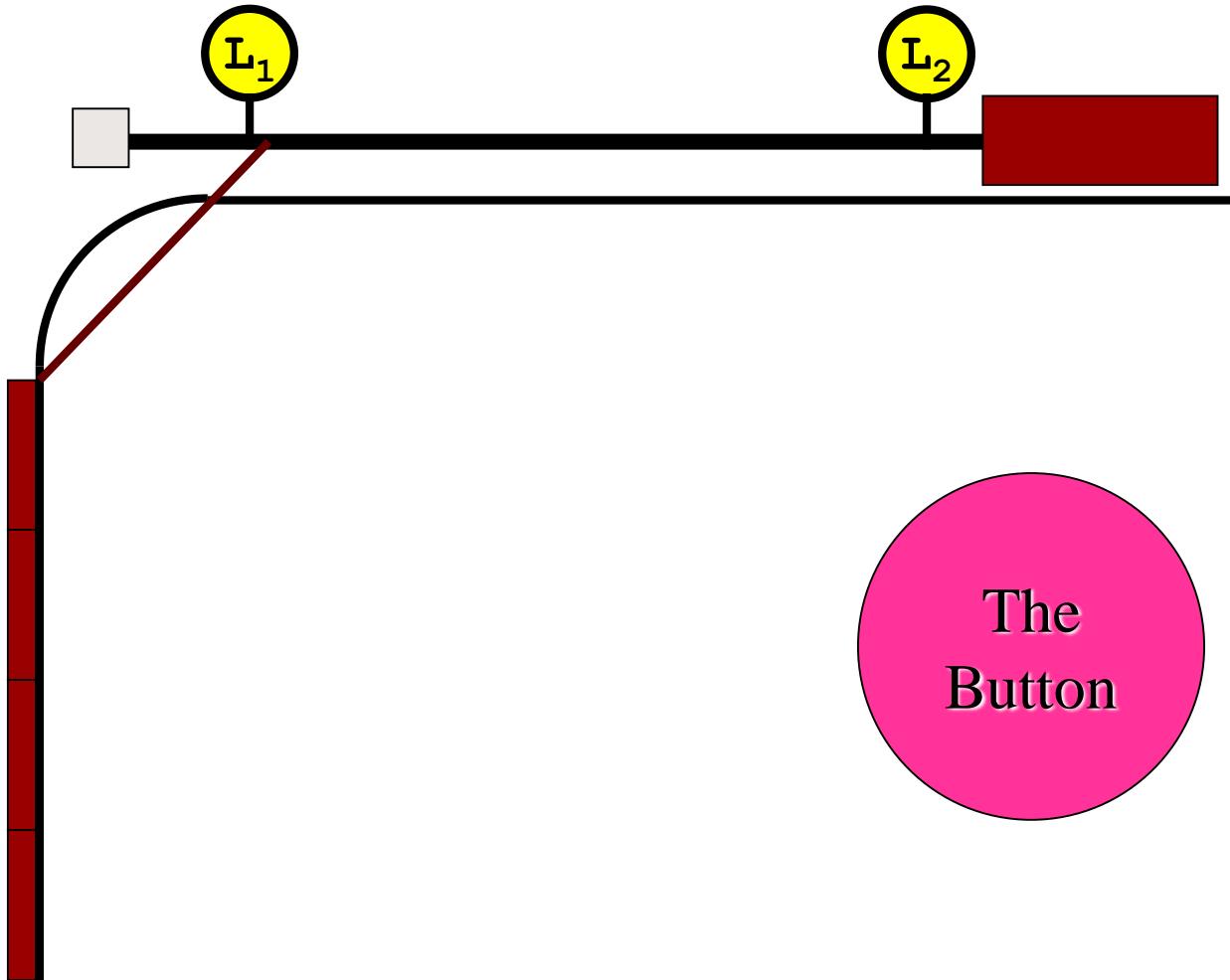


My Garage Door



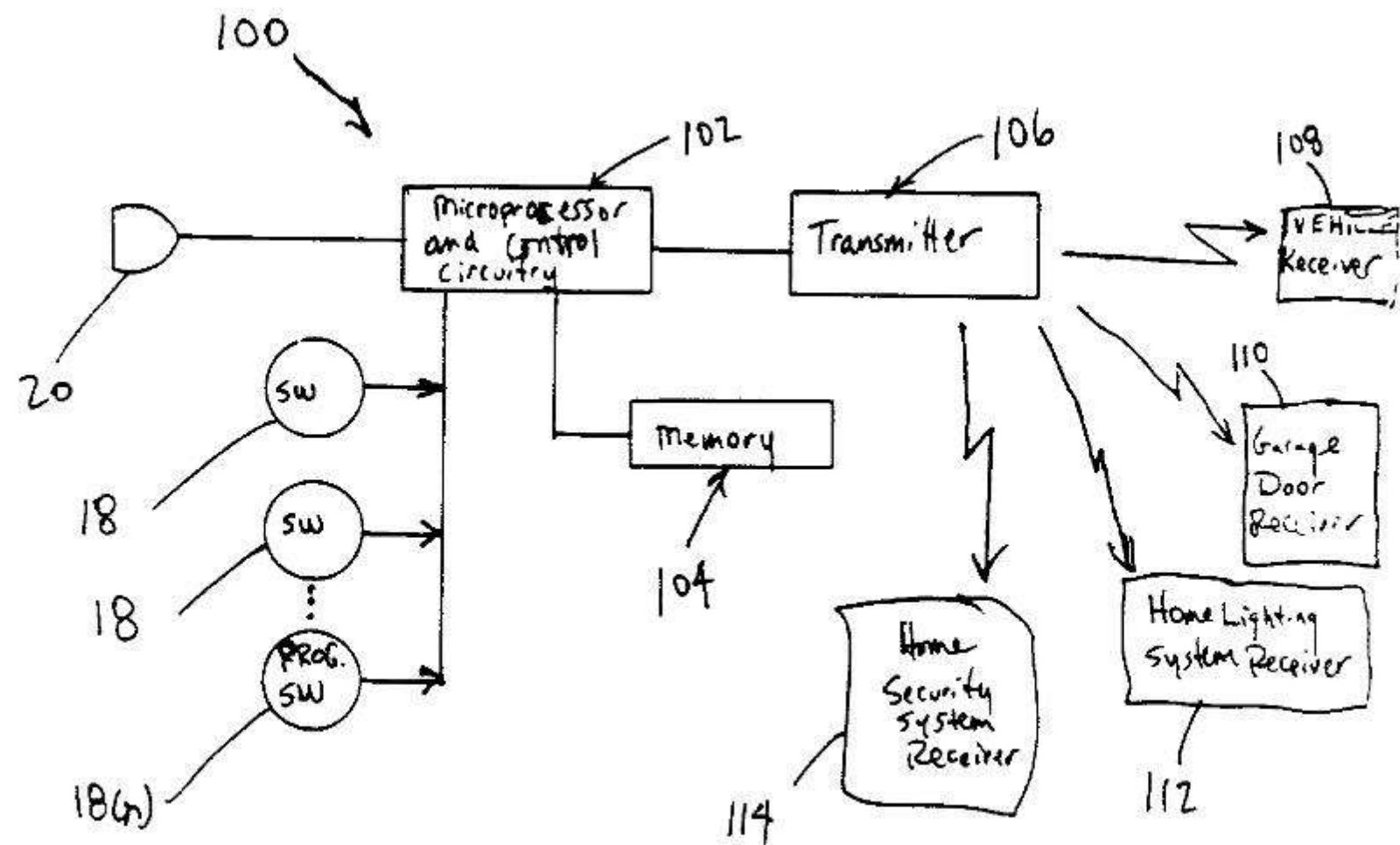
The
Button

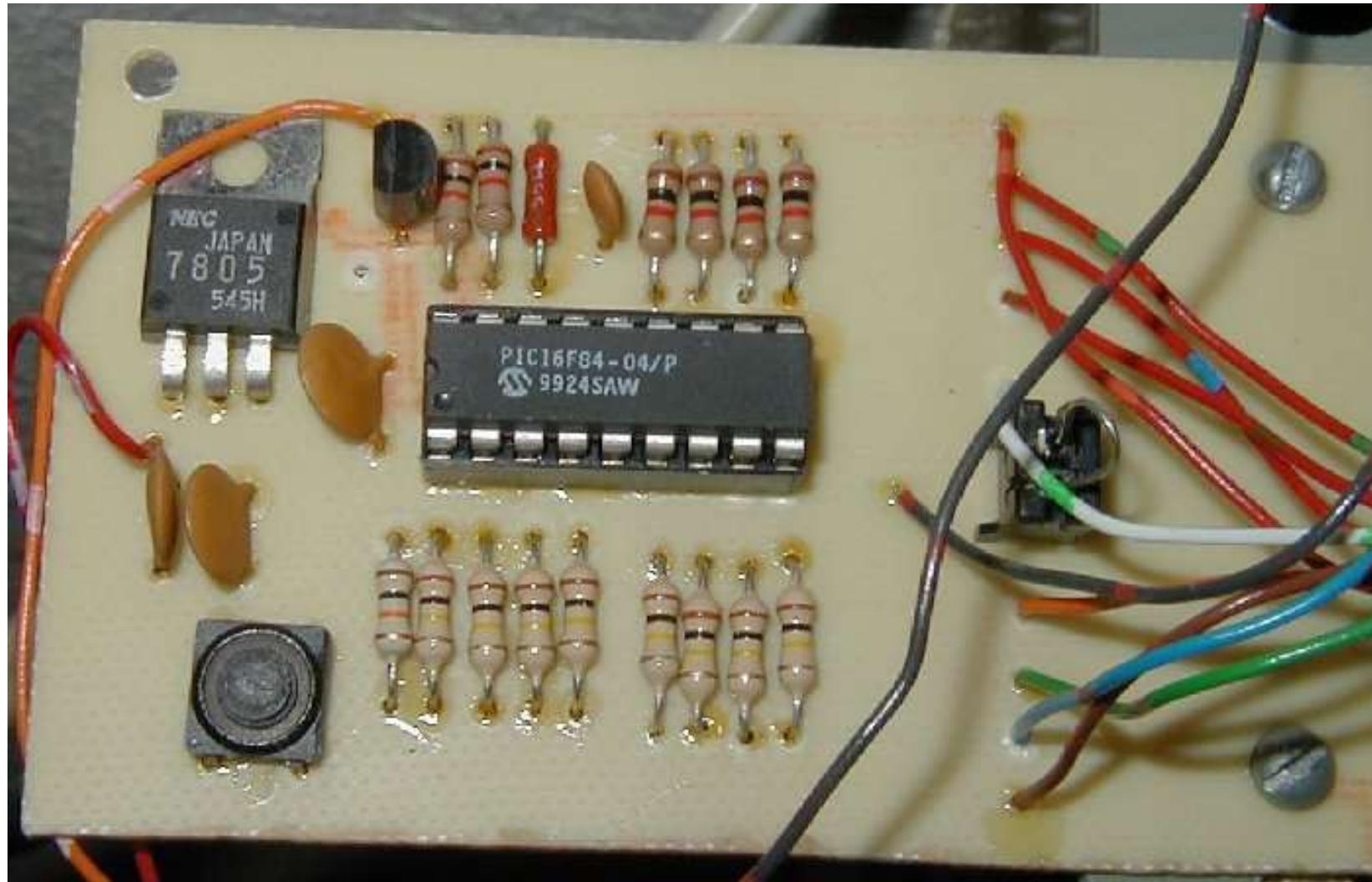
My Garage Door

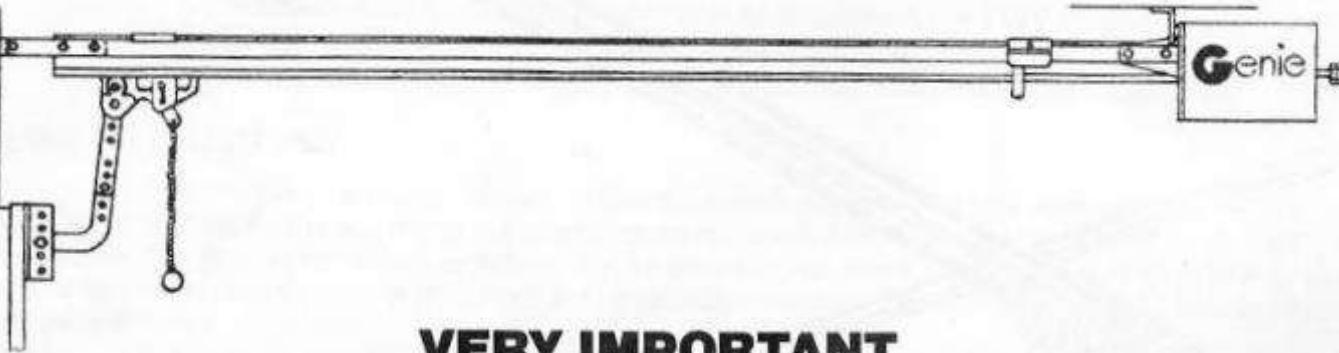


How can we describe the control logic?



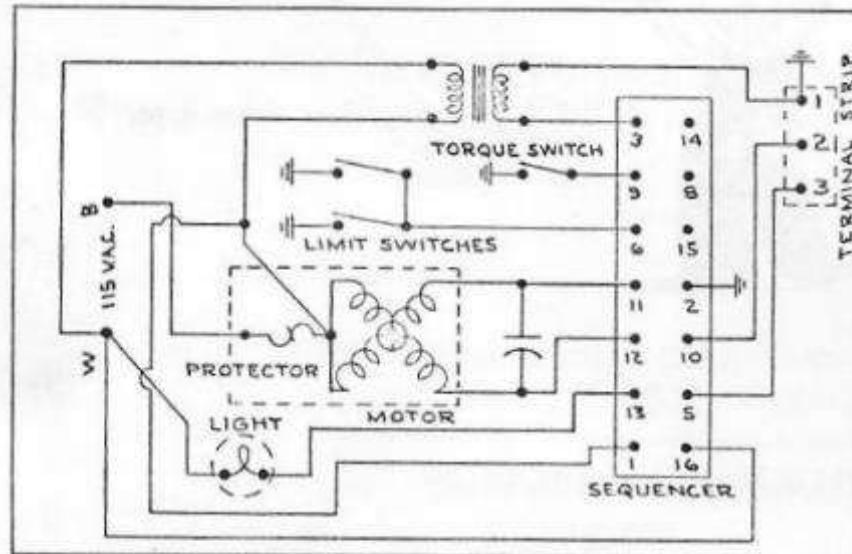




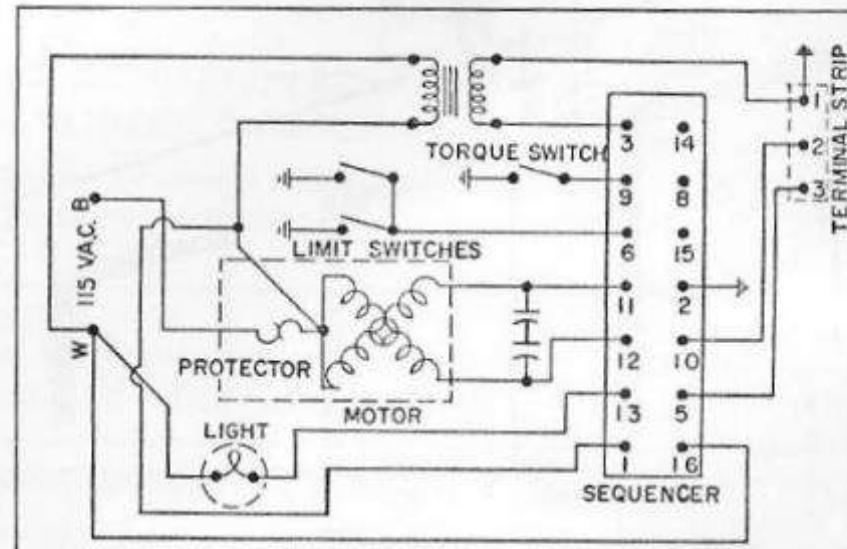


VERY IMPORTANT

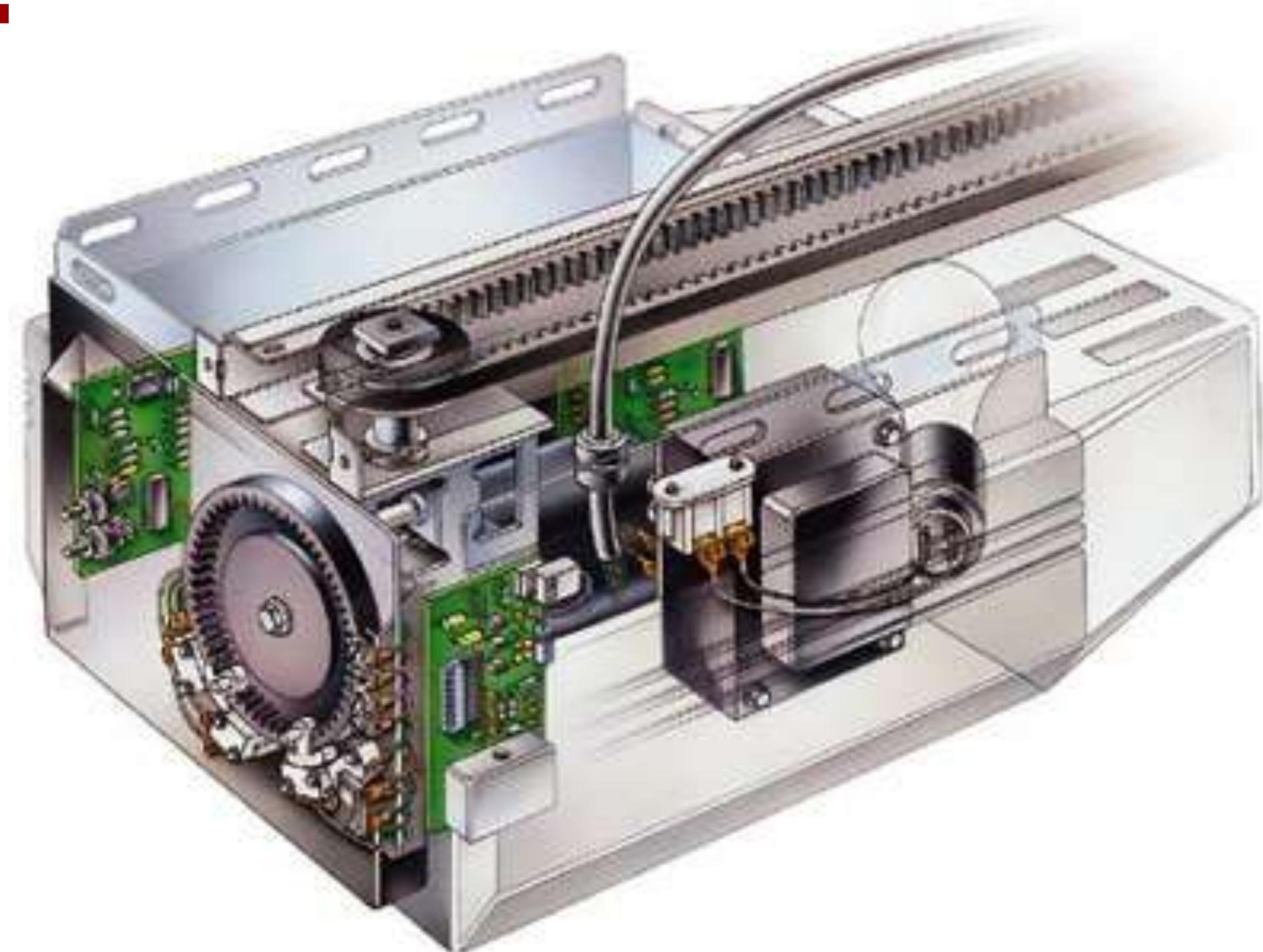
Handles, ropes and any projection on your door not necessary to its operation should be removed when installing a door opener. Failure to remove these items may cause accidental contact or engagement of these projections when door opener is operated. **This is unsafe!** Fixed mechanical stops should be used instead of rope stops on door hardware.



Model 404 wiring diagram



Model 414 wiring diagram



How Can We Describe the Control Logic?

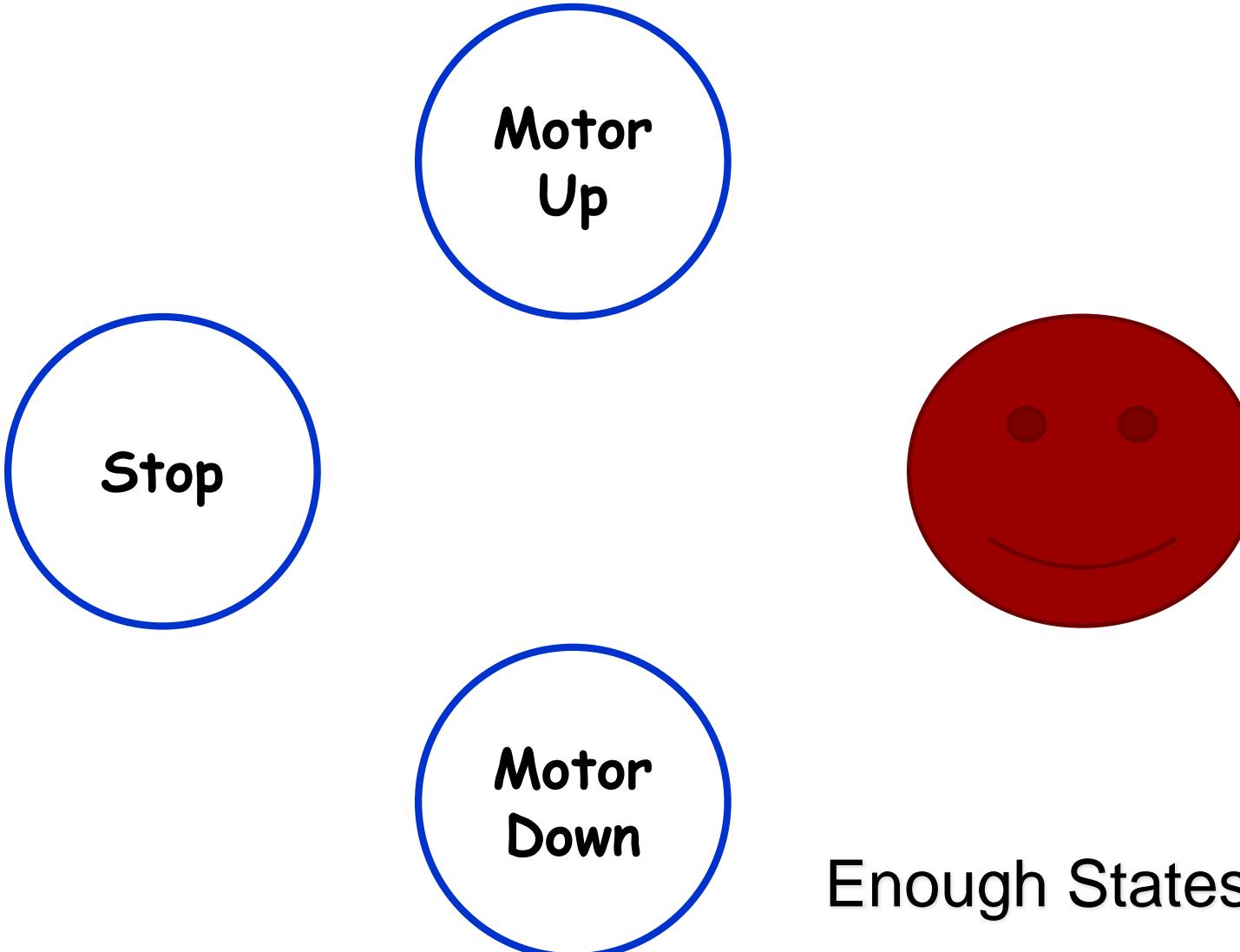
↗ A Finite State Machine
(FSM) Diagram!

So How Will We Do This?

- ↗ Decide how we will encode our states
 - ↗ A binary number will always work
- ↗ Draw a state diagram including
 - ↗ Labeled states (encoding)
 - ↗ Outputs during the state
 - ↗ Arcs with conditions for state changes
- ↗ Create truth tables with
 - ↗ Inputs: **Circuit inputs** and **Current state**
 - ↗ Output: **Circuit outputs** and **Next state**
 - ↗ Fill in the outputs for all combinations based on the state diagram
- ↗ Implement a combinational circuit for each output using the truth table

Start

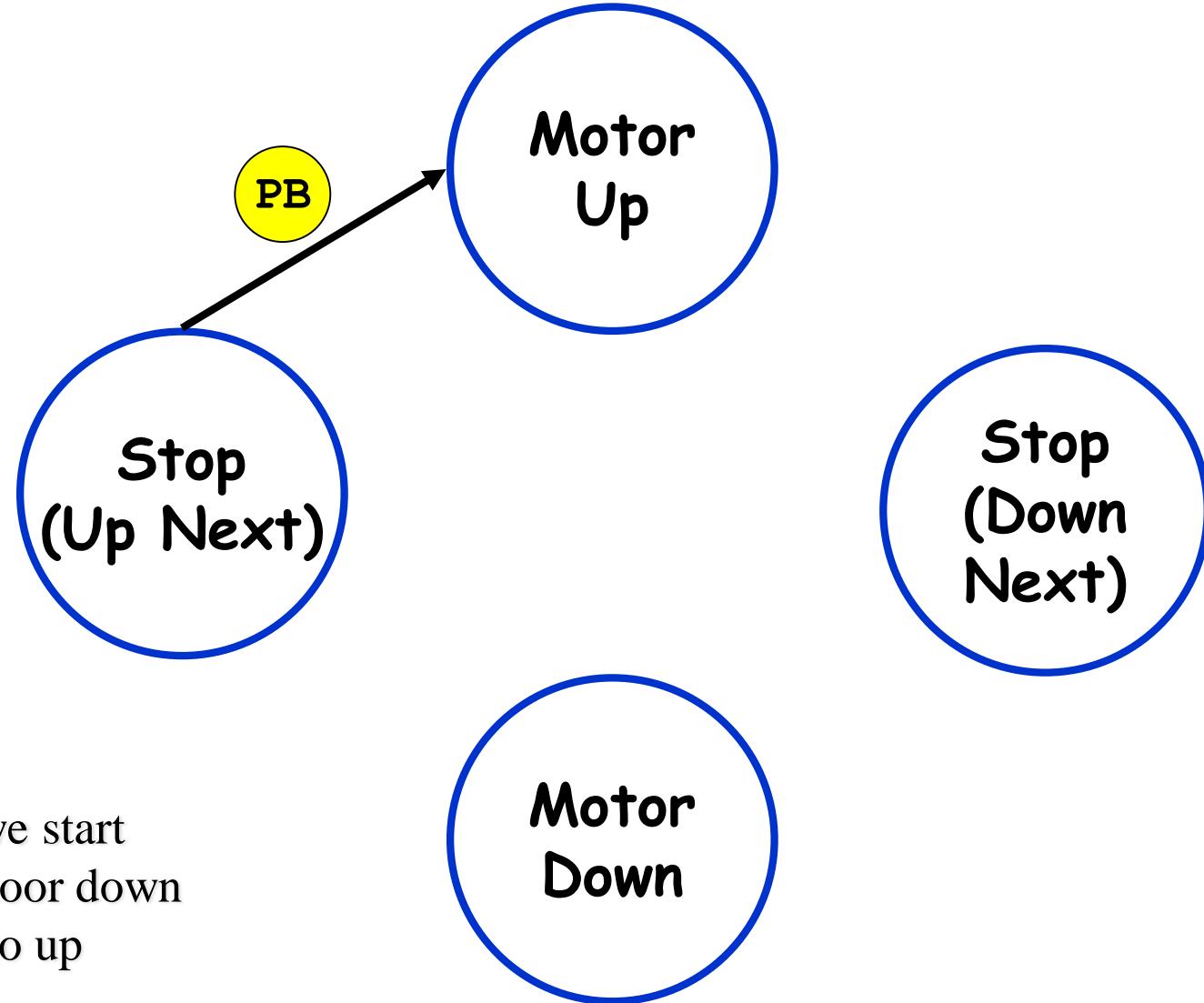
State Machine



Enough States?

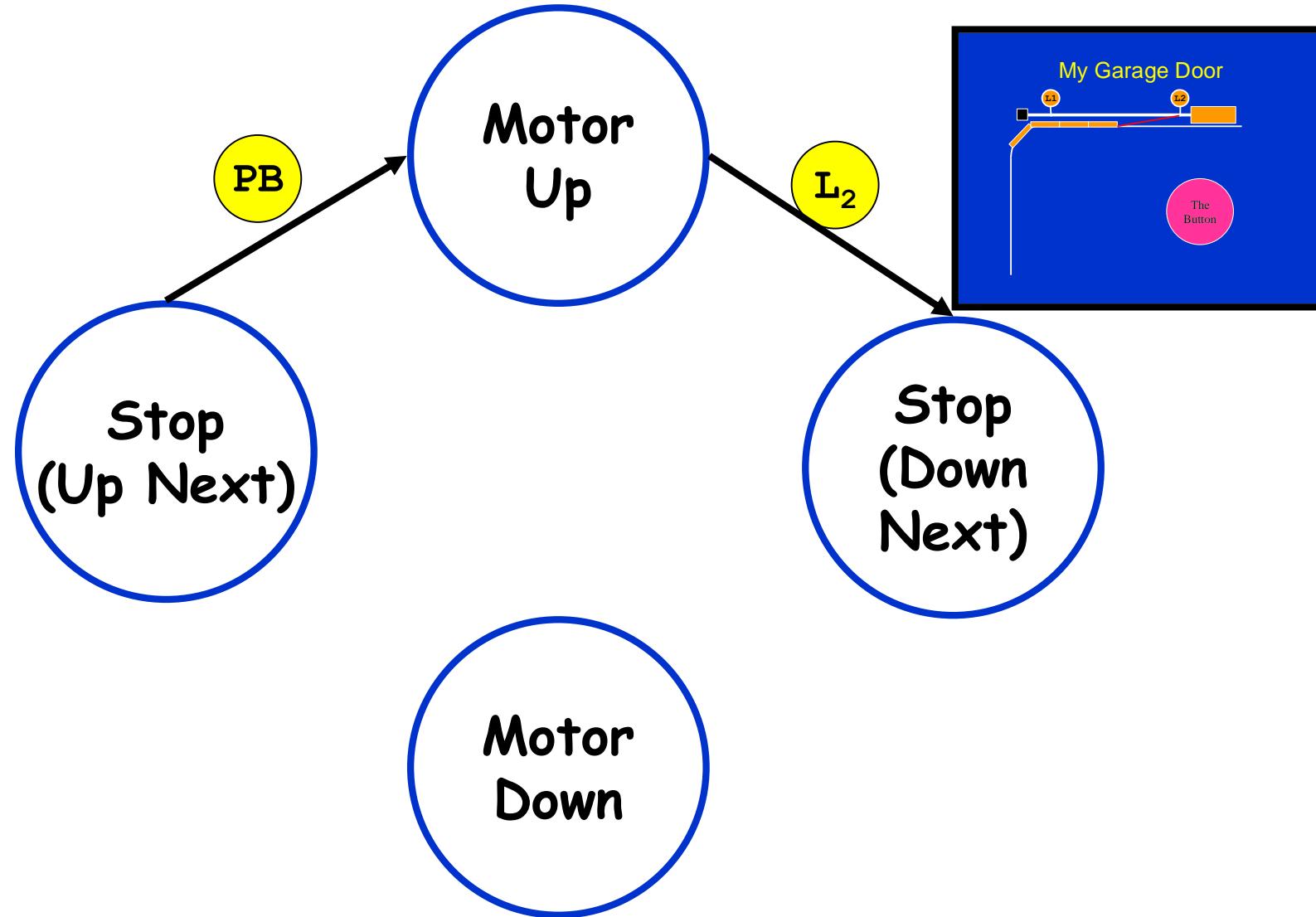
Start

State Machine



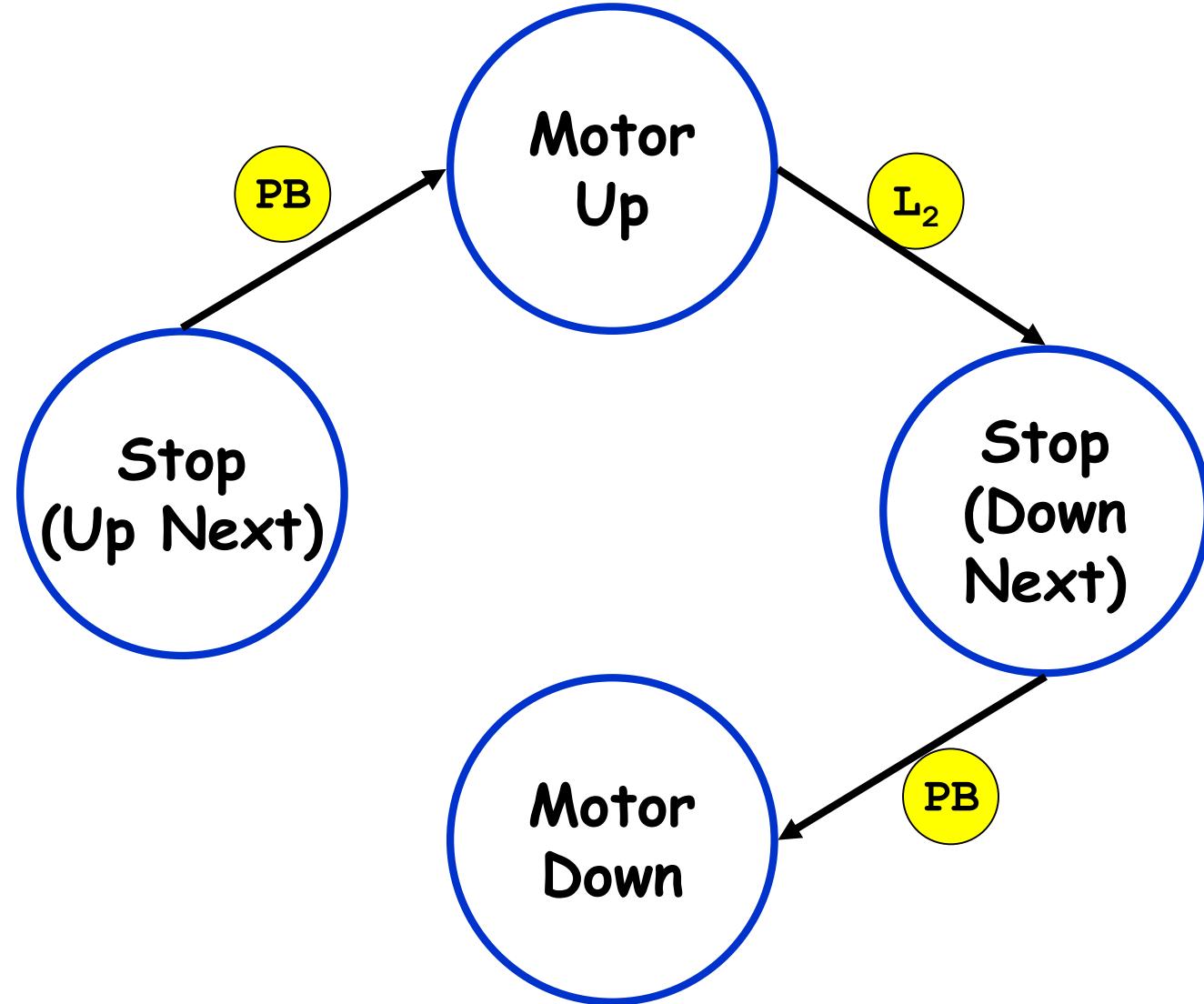
Assume we start
with the door down
ready to go up

State Machine



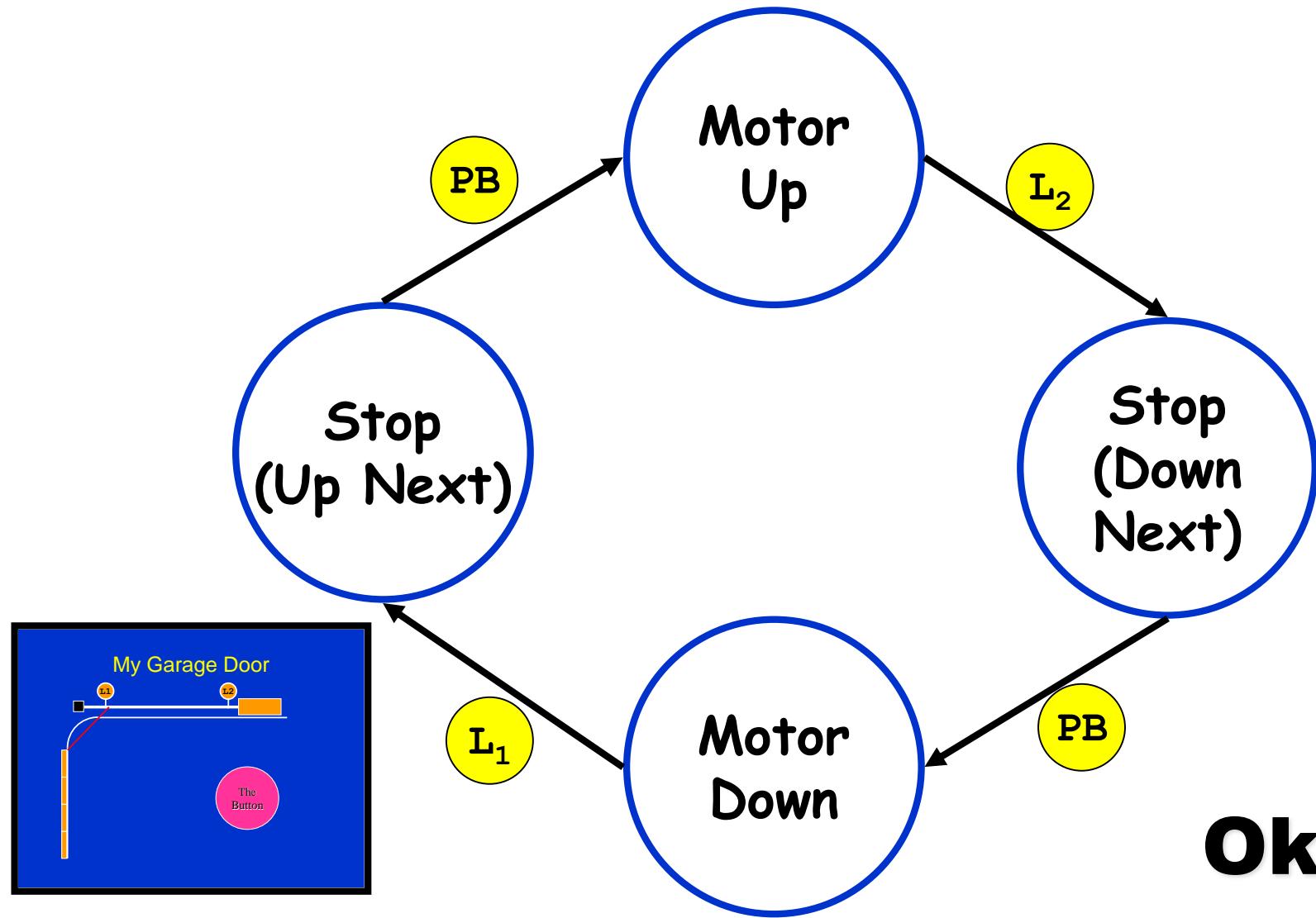
Start

State Machine



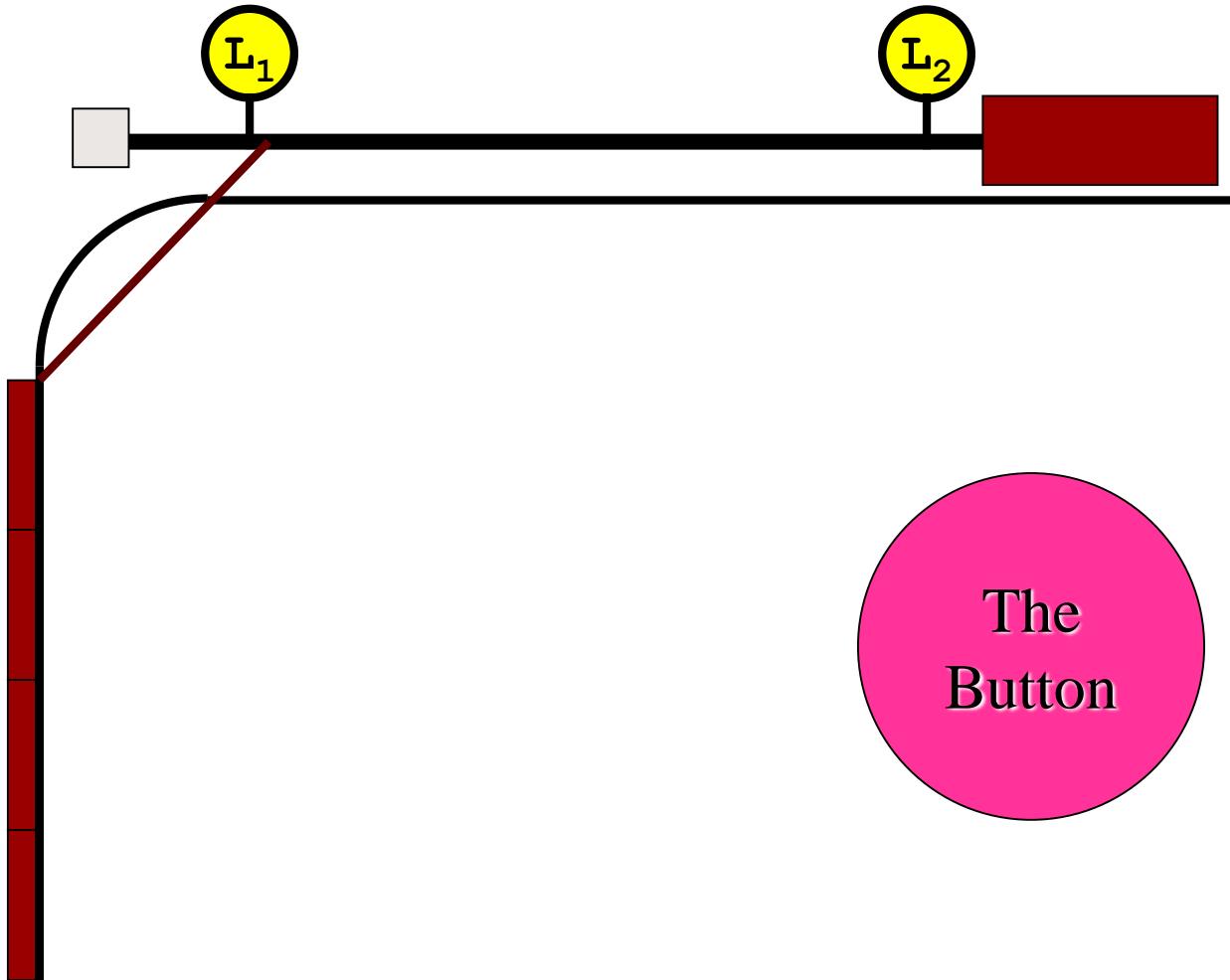
Start

State Machine

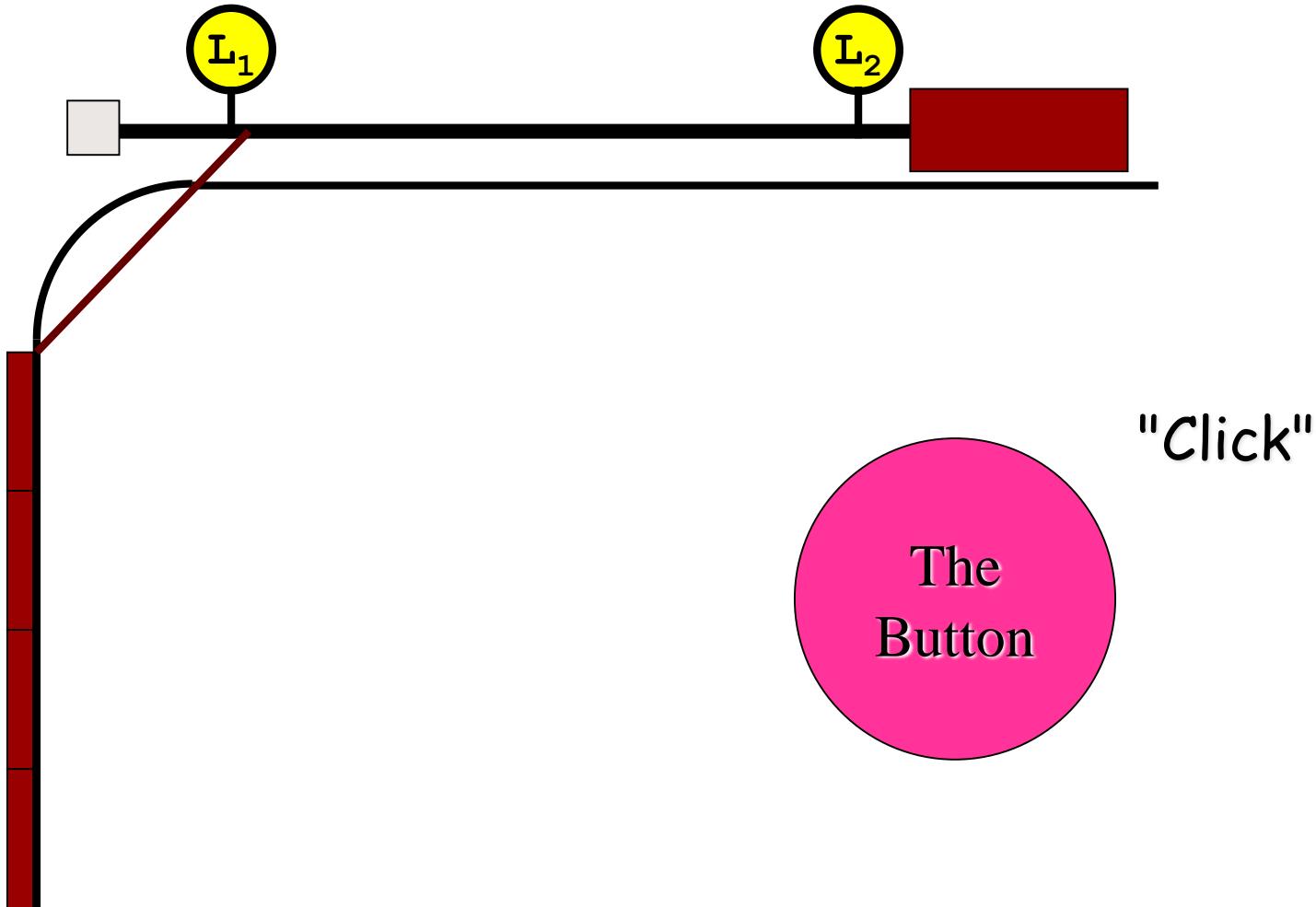


Okay?

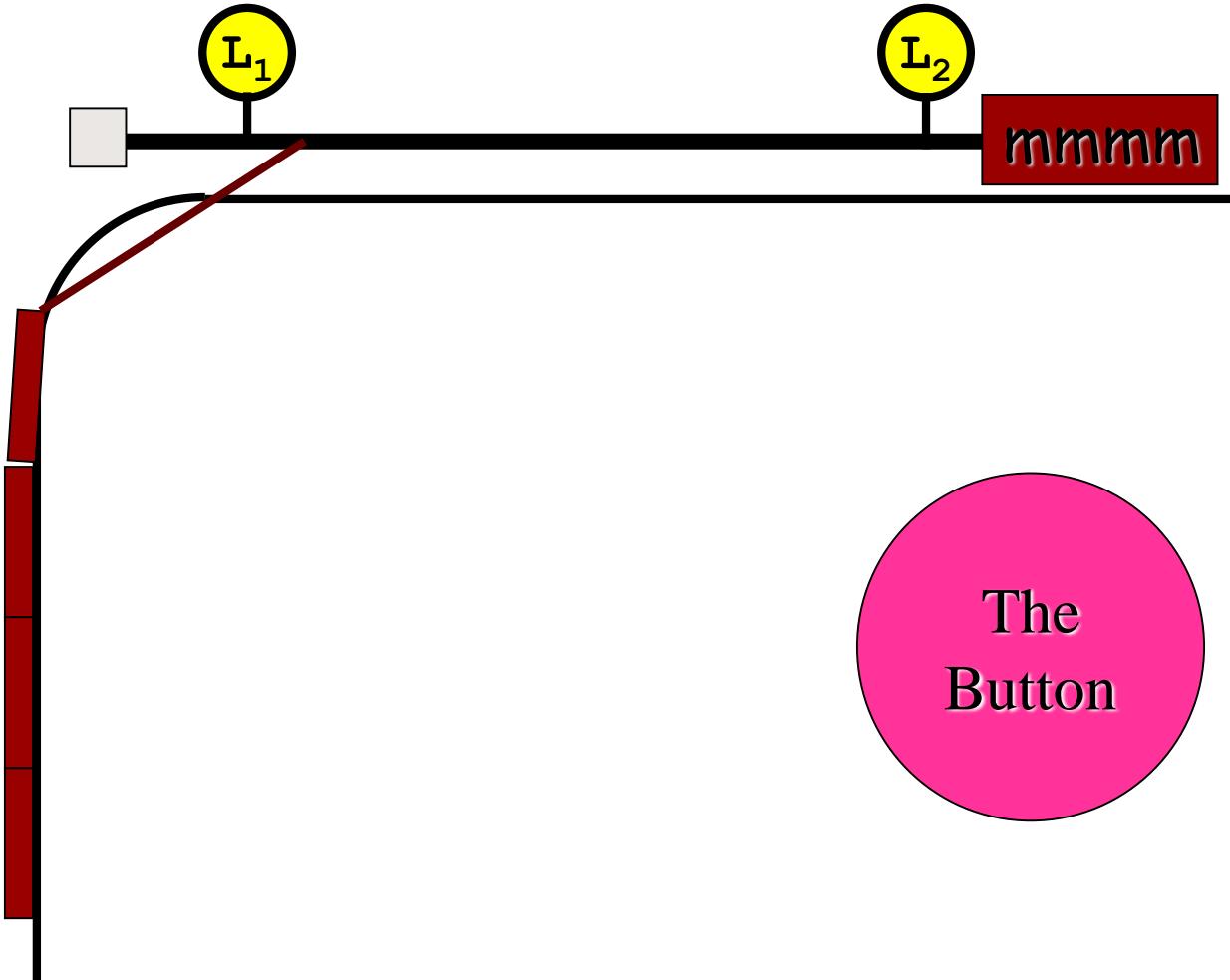
My Garage Door



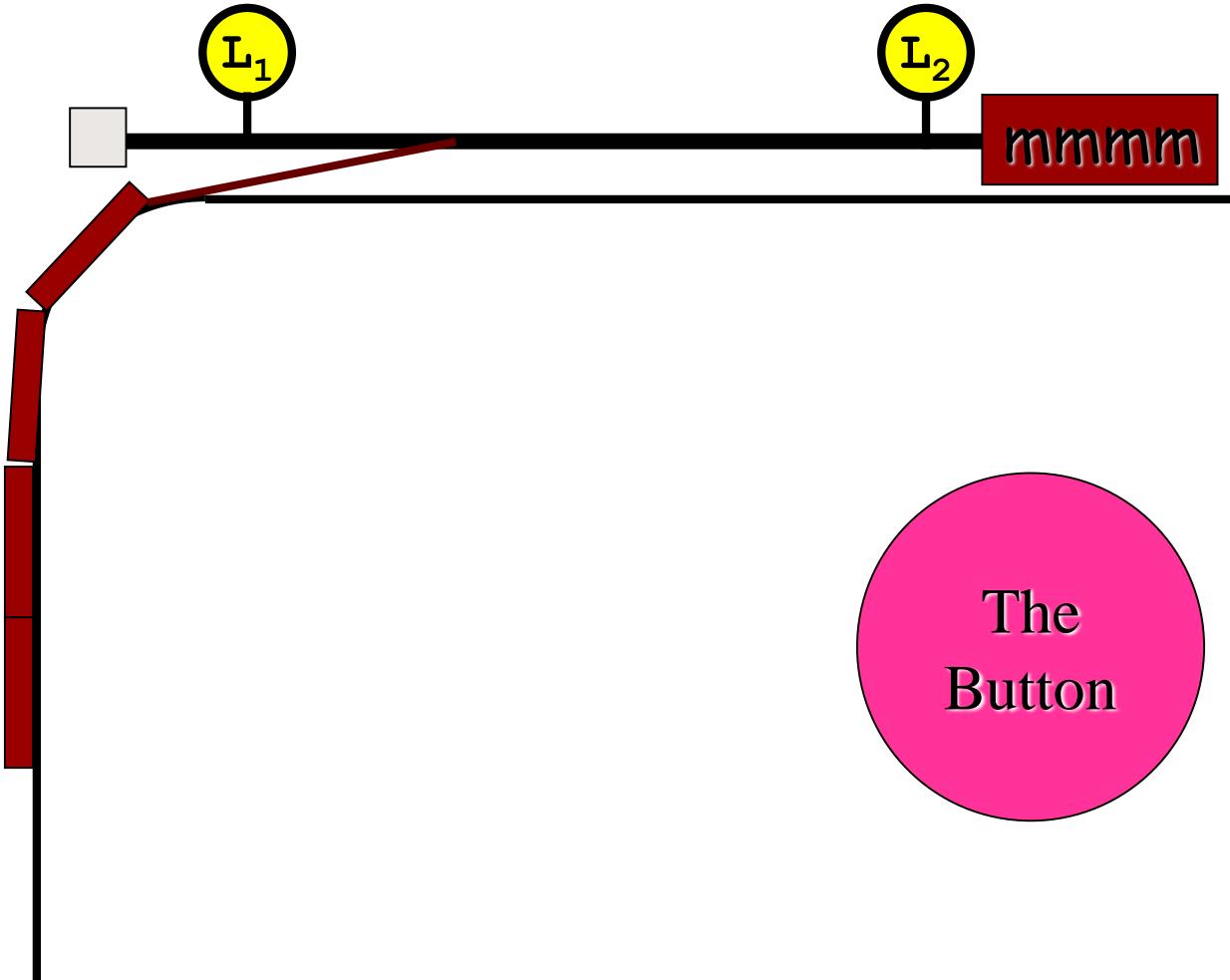
My Garage Door



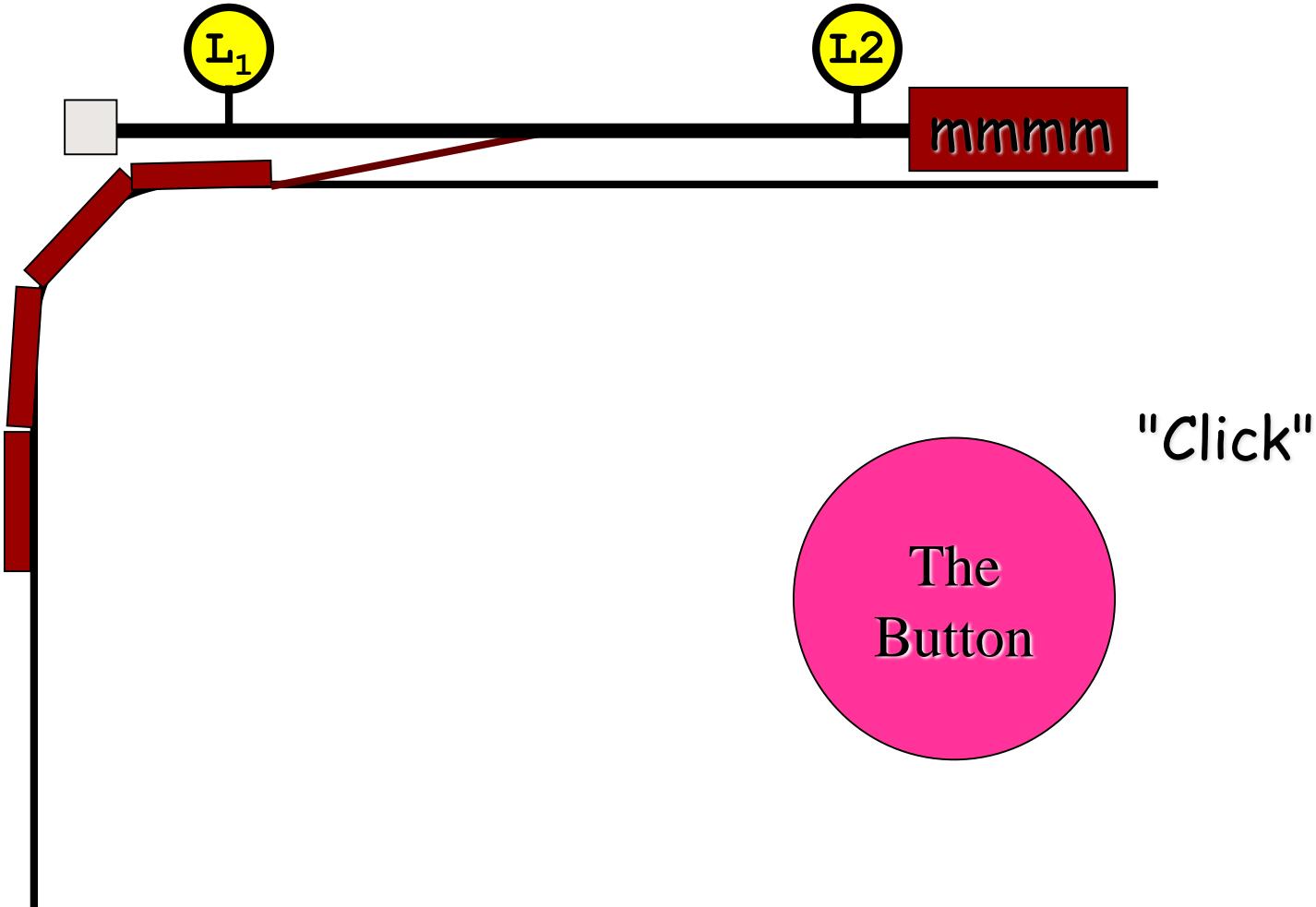
My Garage Door



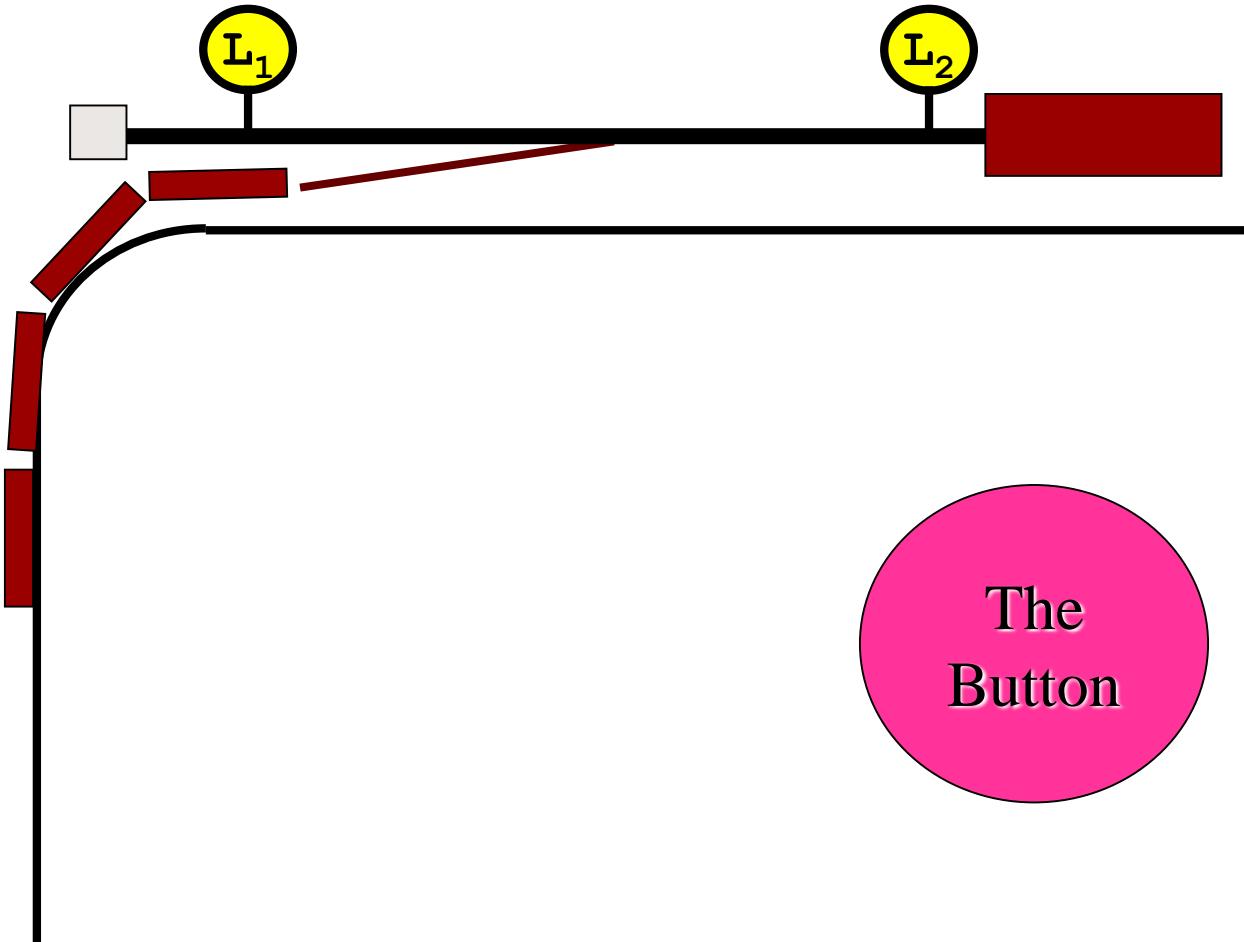
My Garage Door



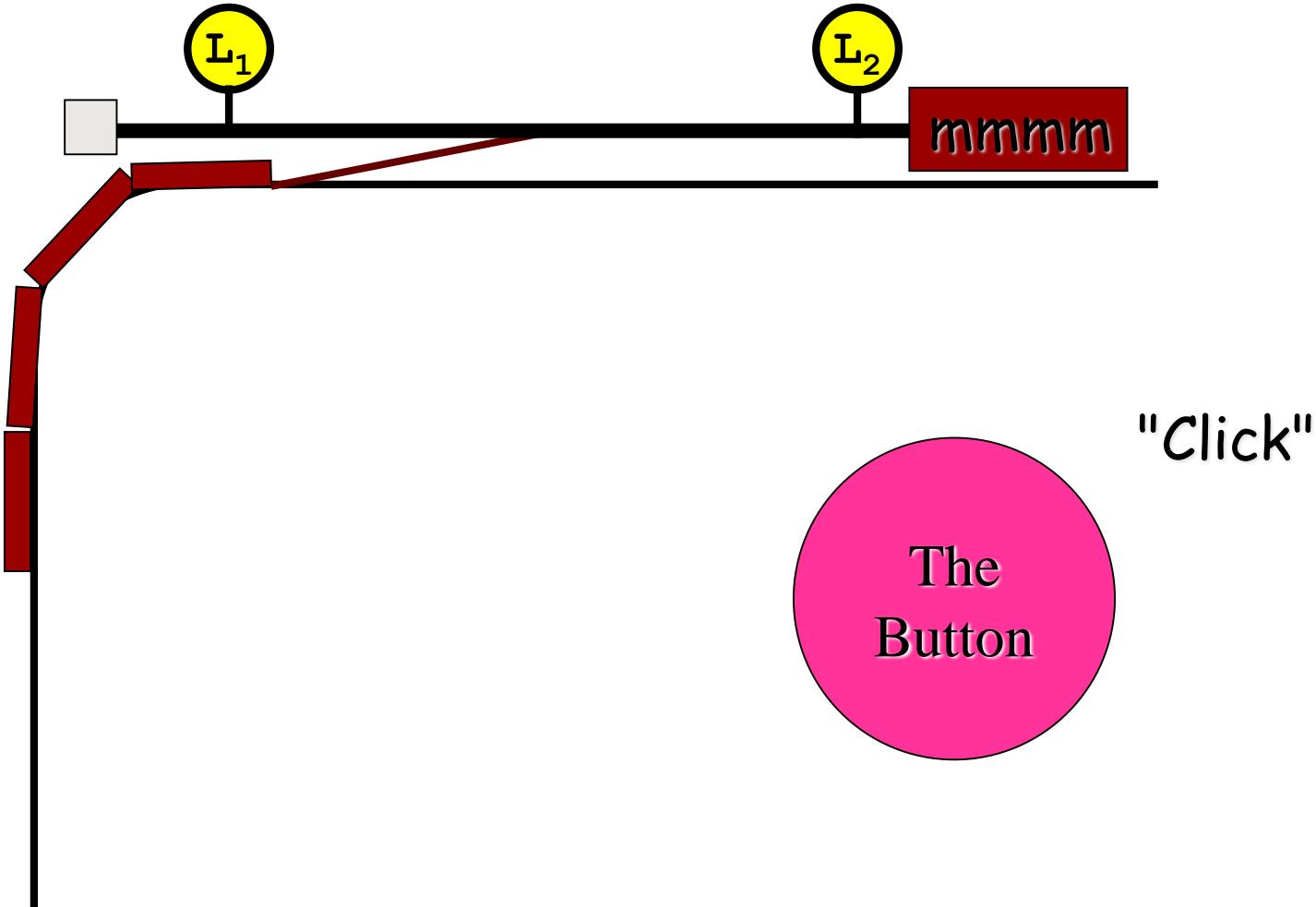
My Garage Door



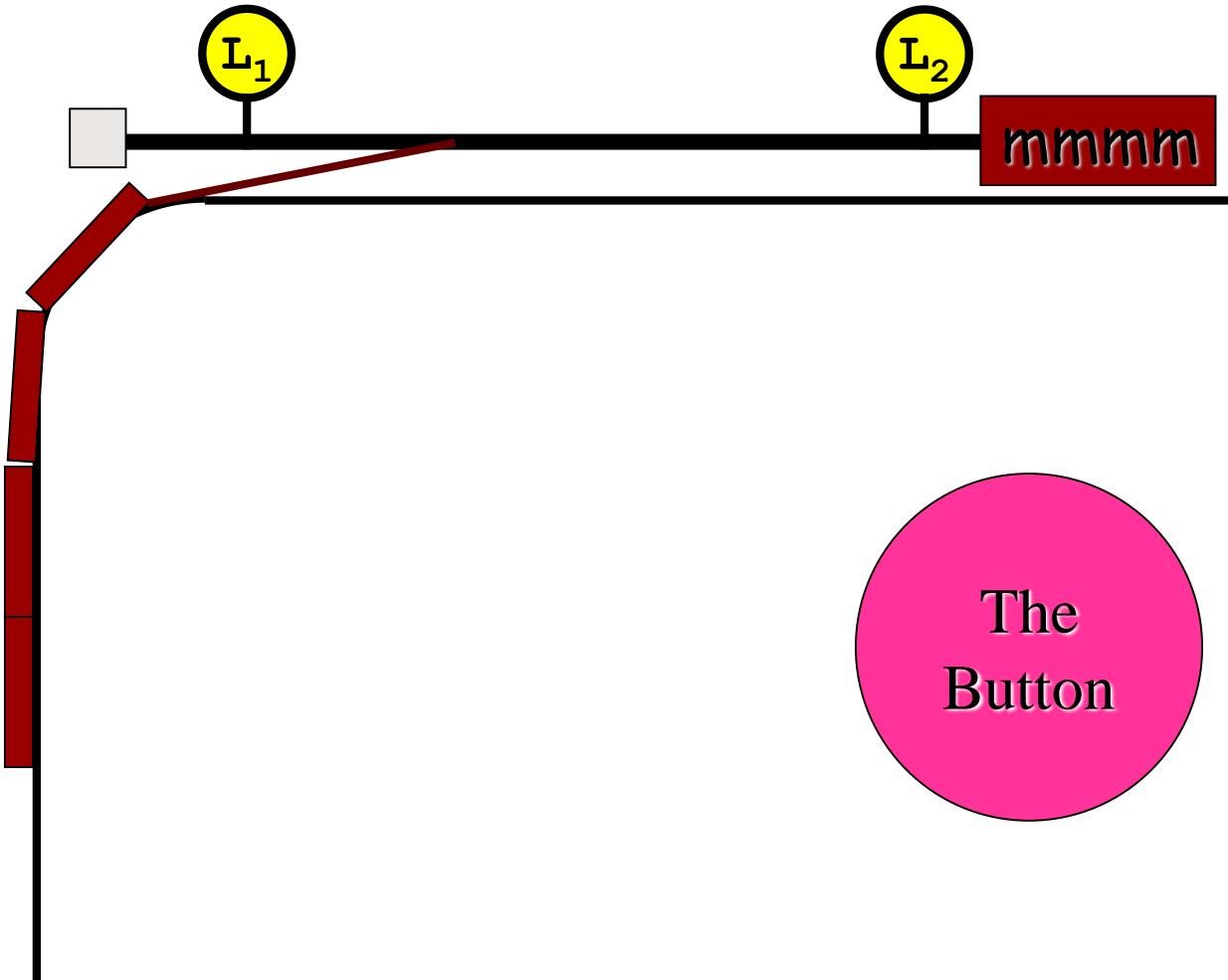
My Garage Door



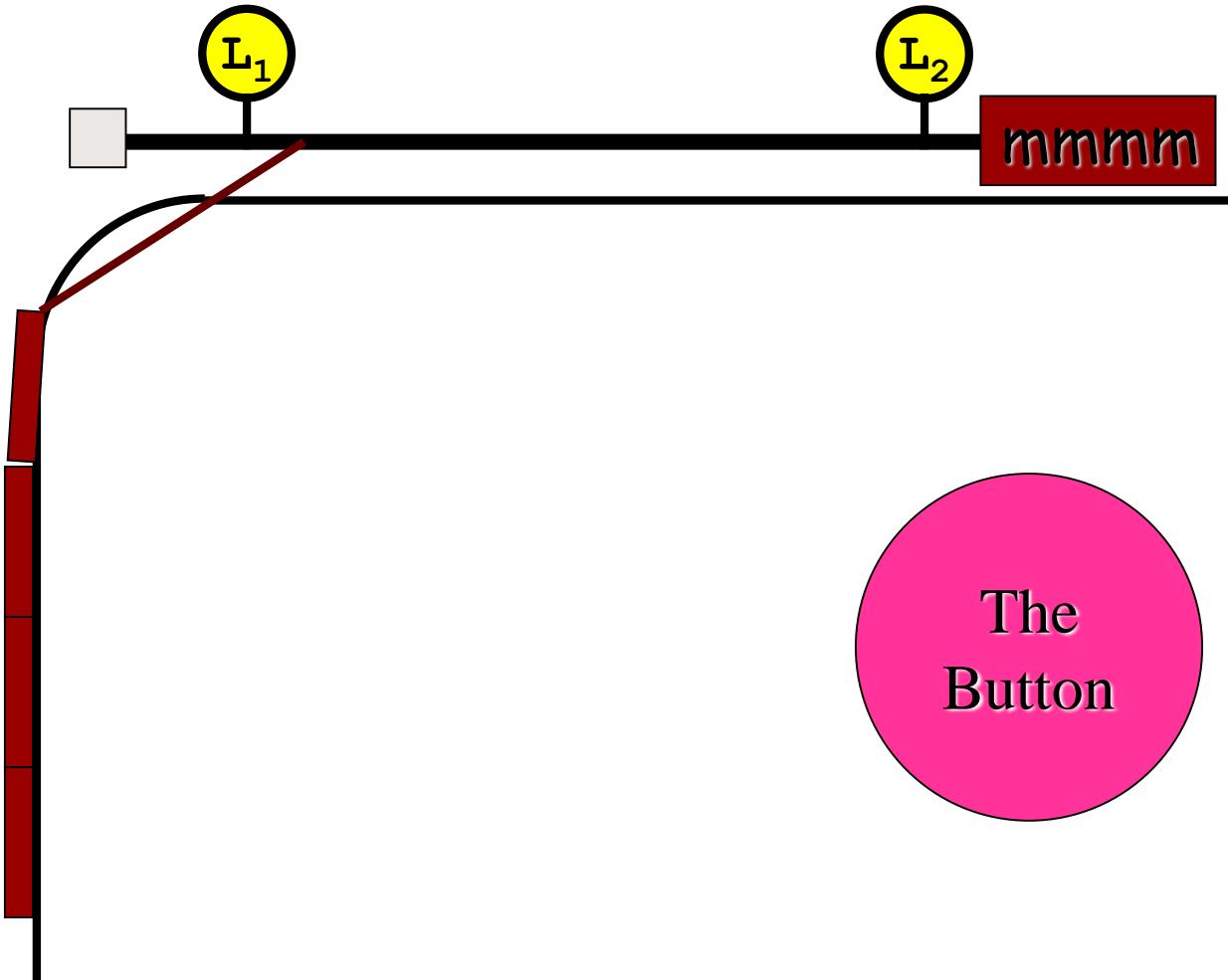
My Garage Door



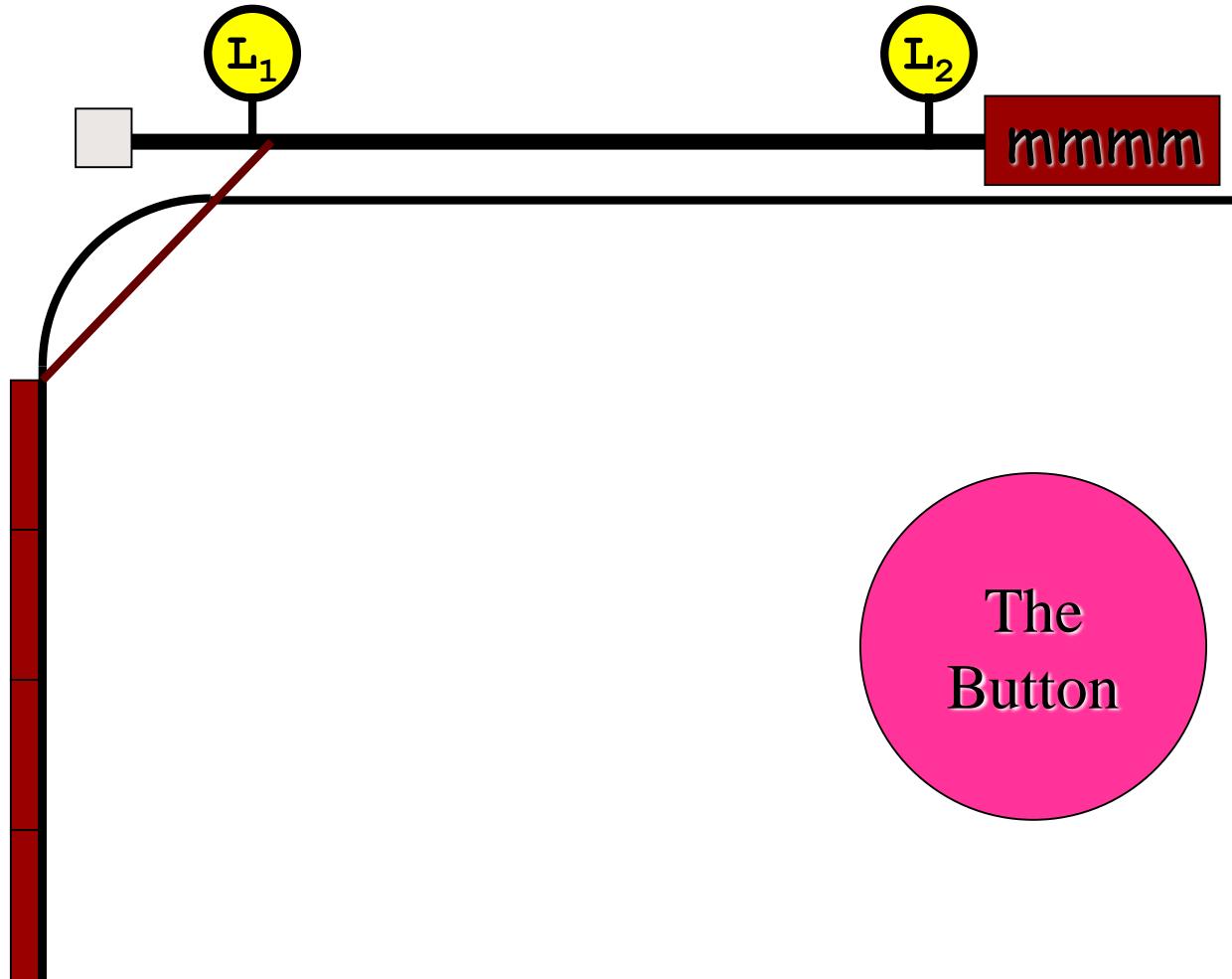
My Garage Door



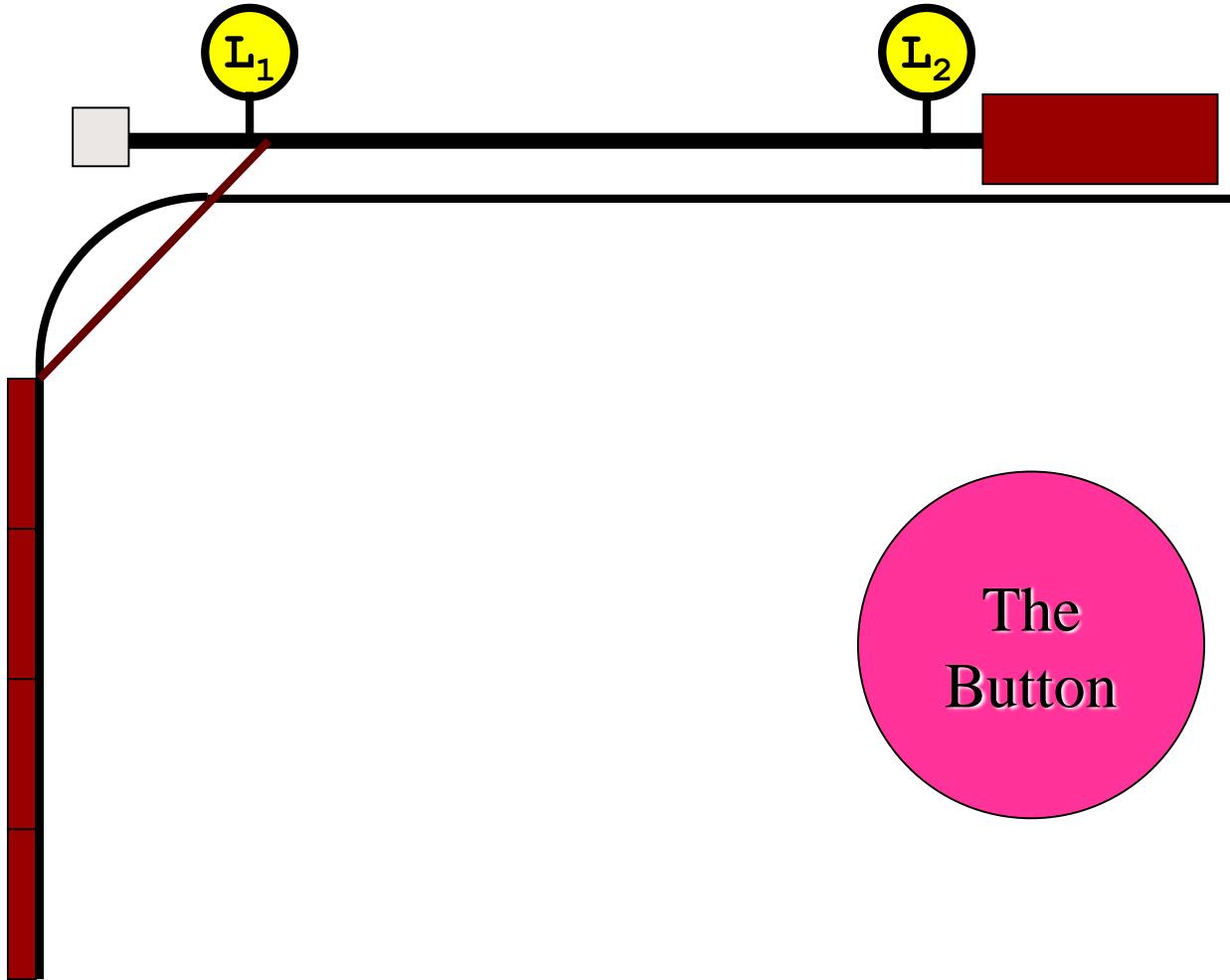
My Garage Door



My Garage Door

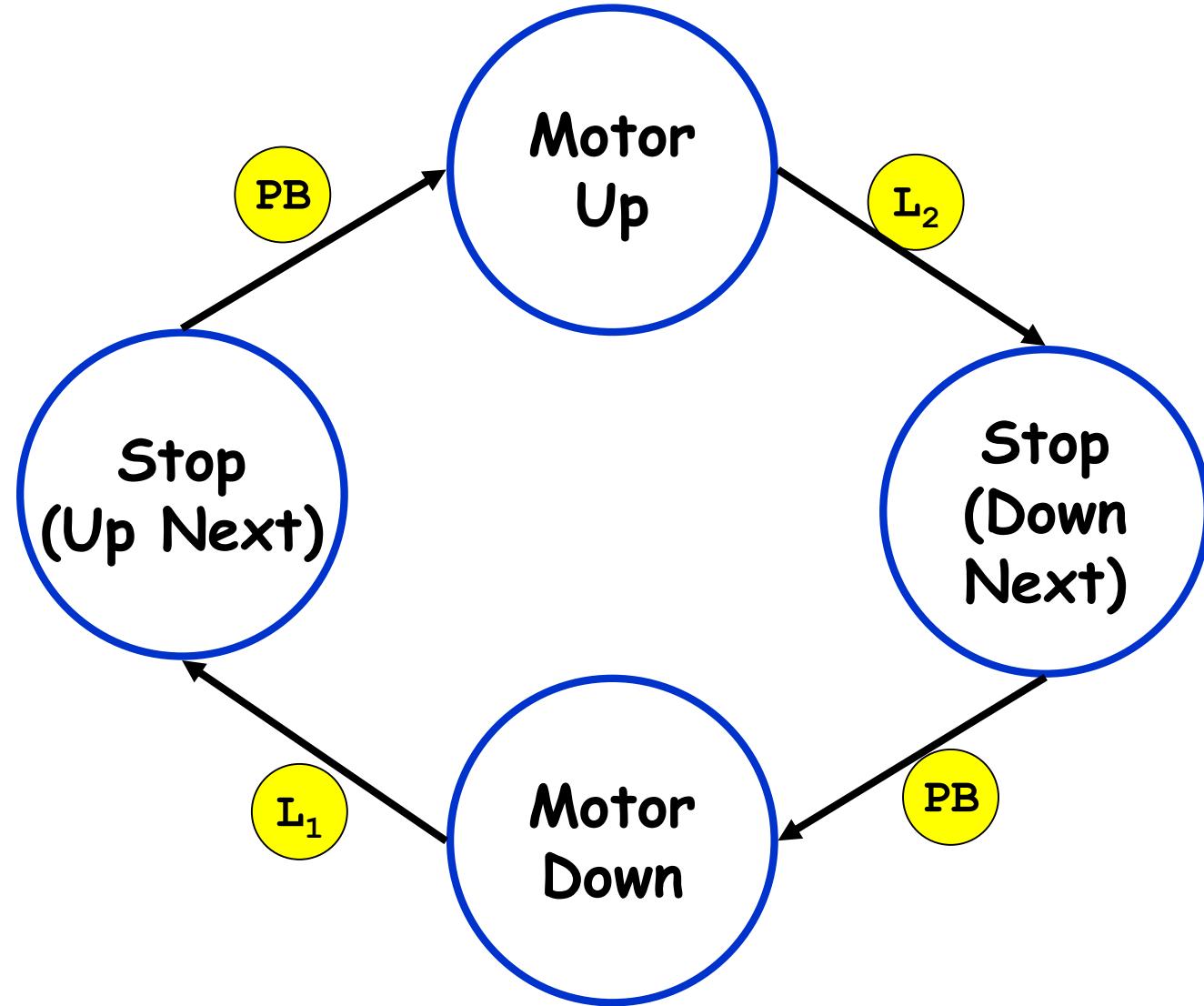


My Garage Door



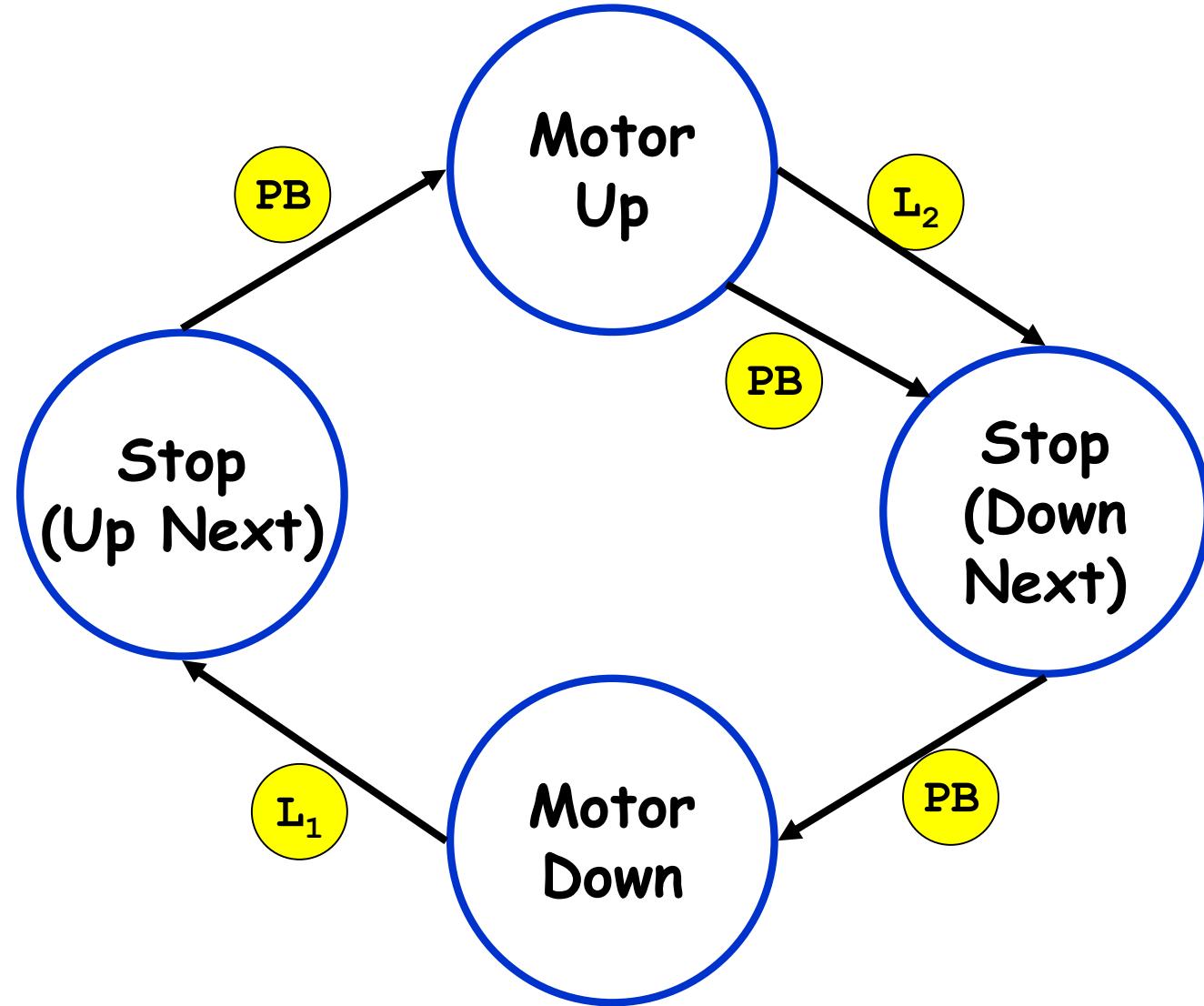
Start

State Machine



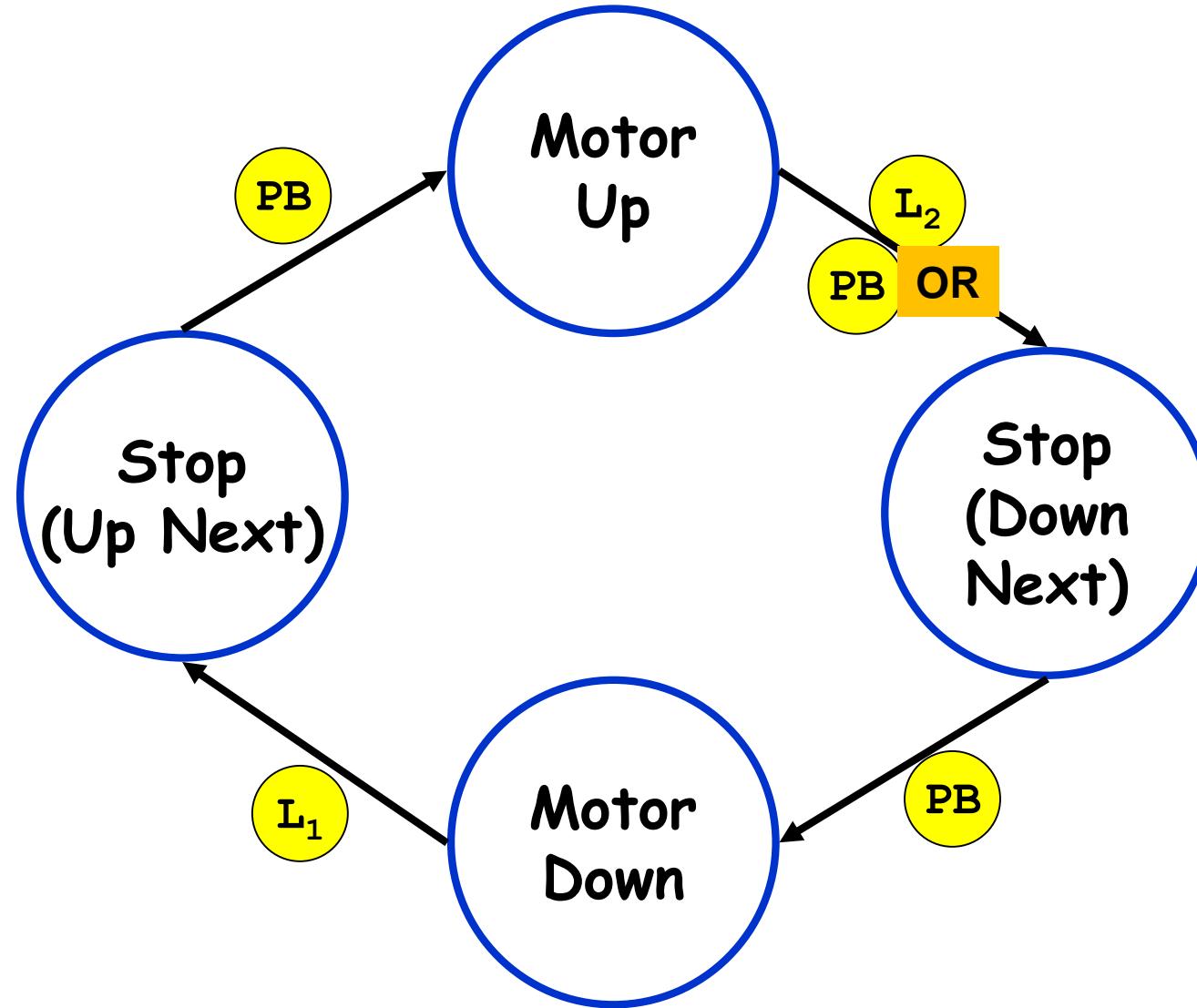
Start

State Machine



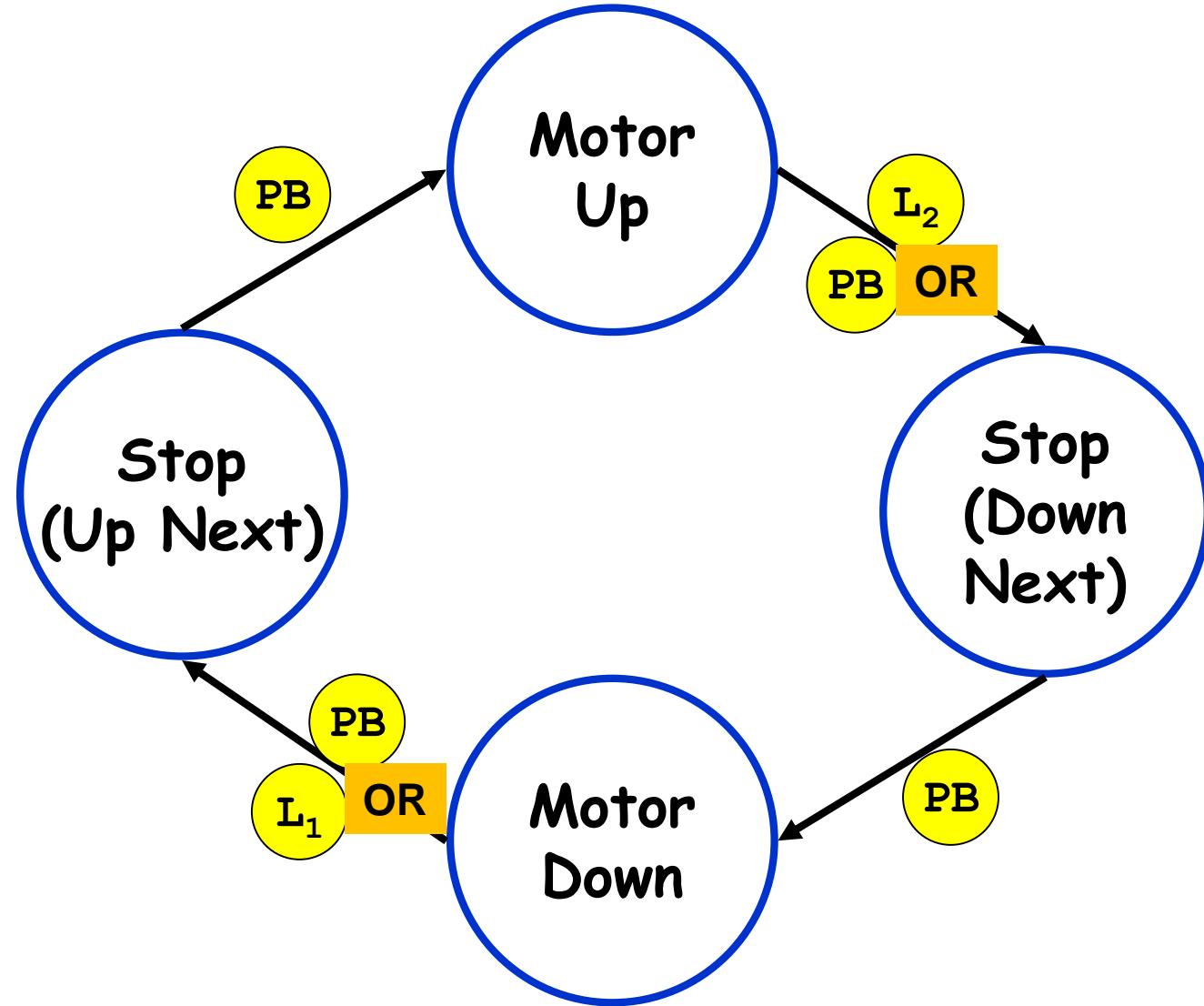
Start

State Machine



Start

State Machine



Suppose we wish to build a circuit to control our garage door?

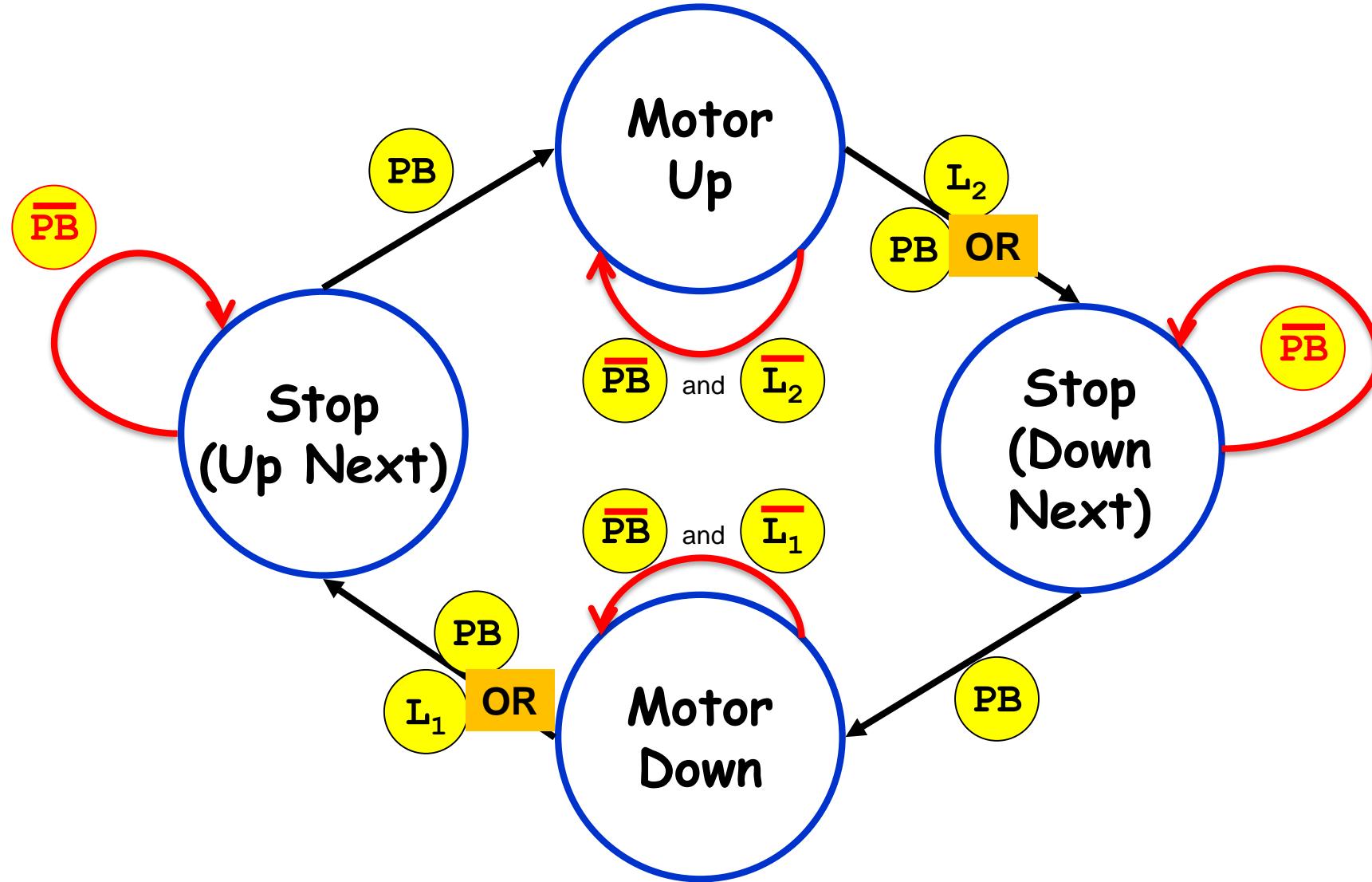
- ↗ What will we need?

What do we need?

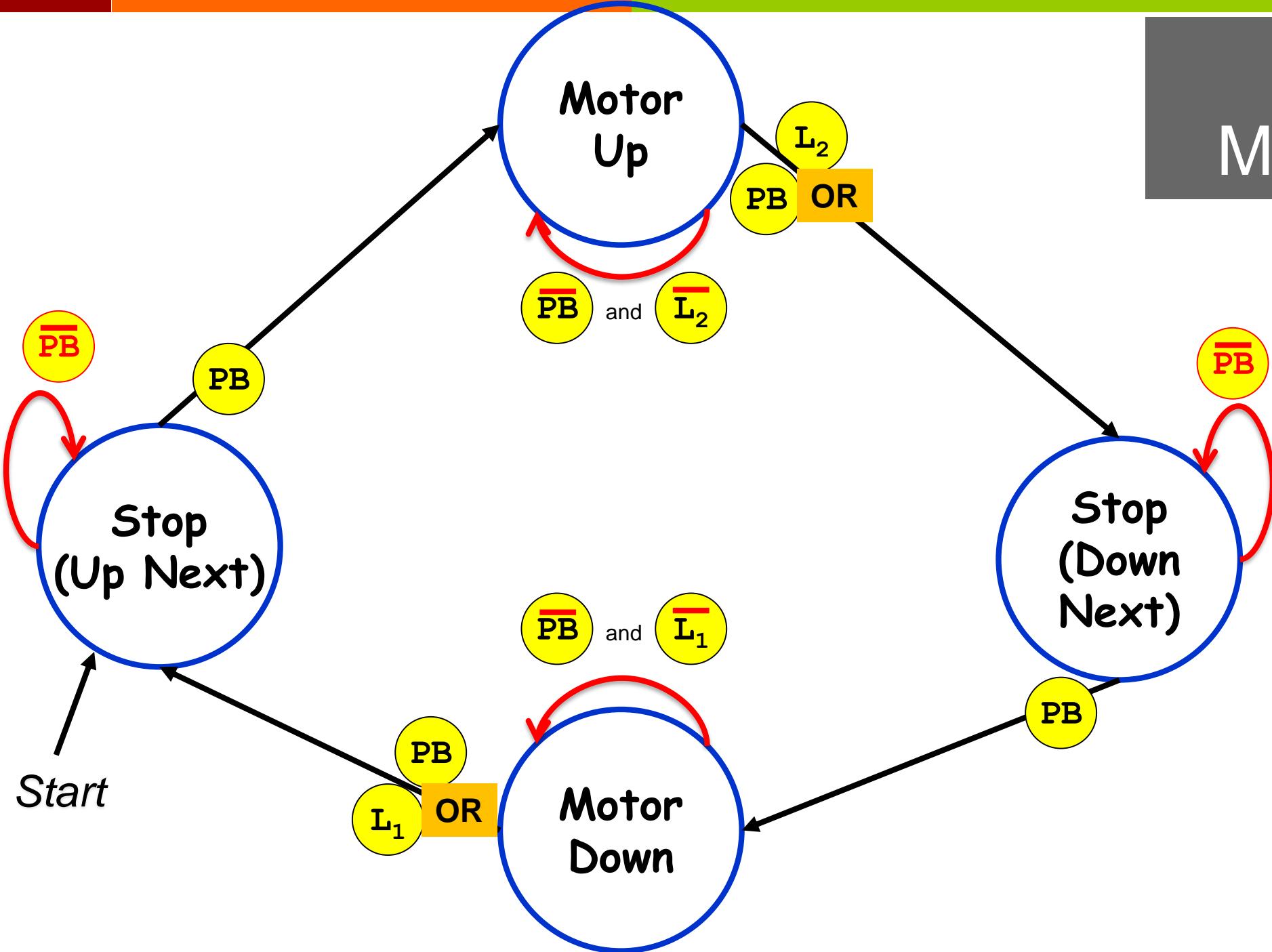
- ↗ Something to hold state
- ↗ Combinational logic
- ↗ A clock
- ↗ Reset button
- ↗ Limit switches
- ↗ The Pushbutton
- ↗ Handle being in no state! i.e. Startup
- ↗ What keeps you in a state?

Start

State Machine

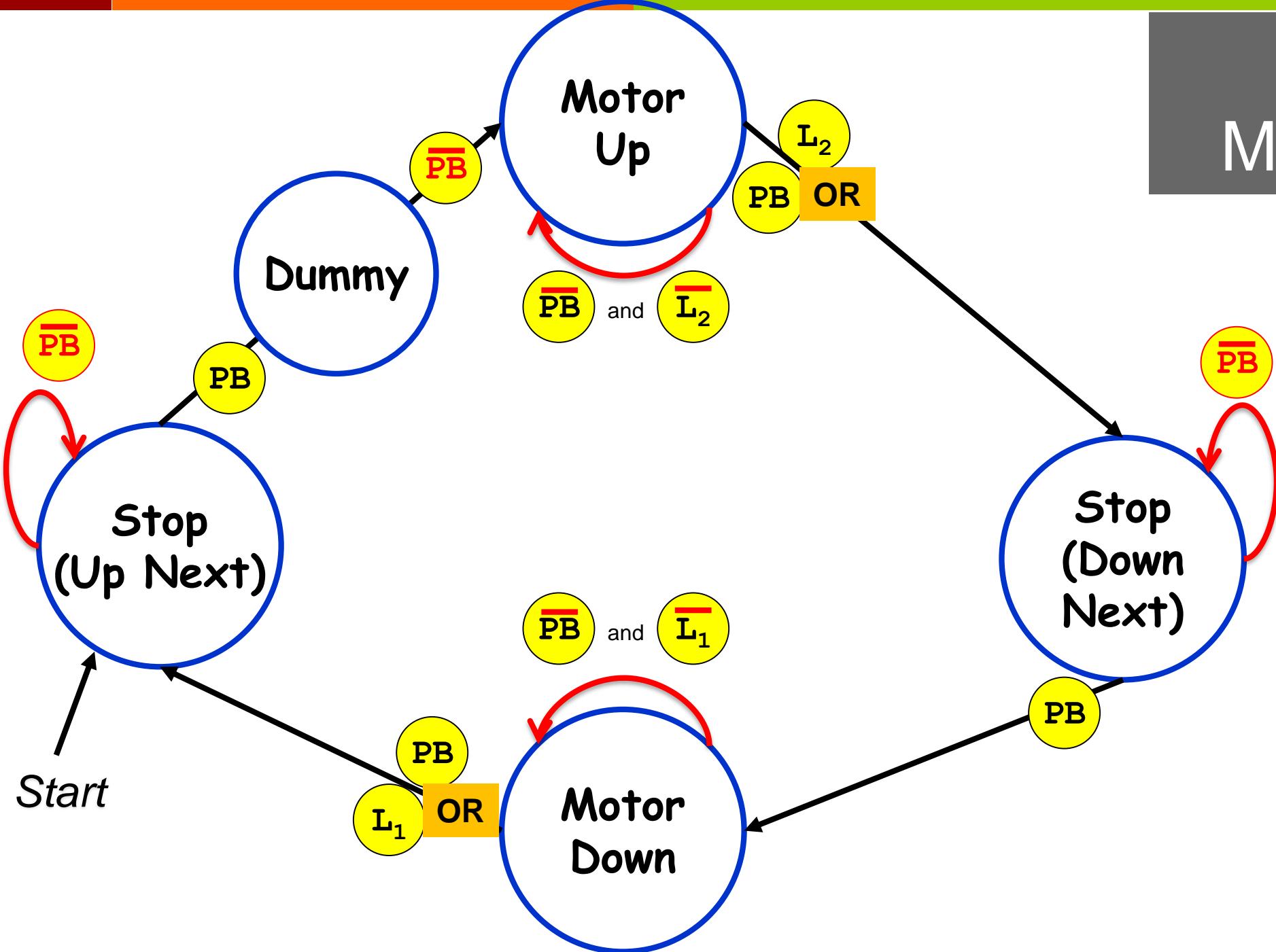


State Machine

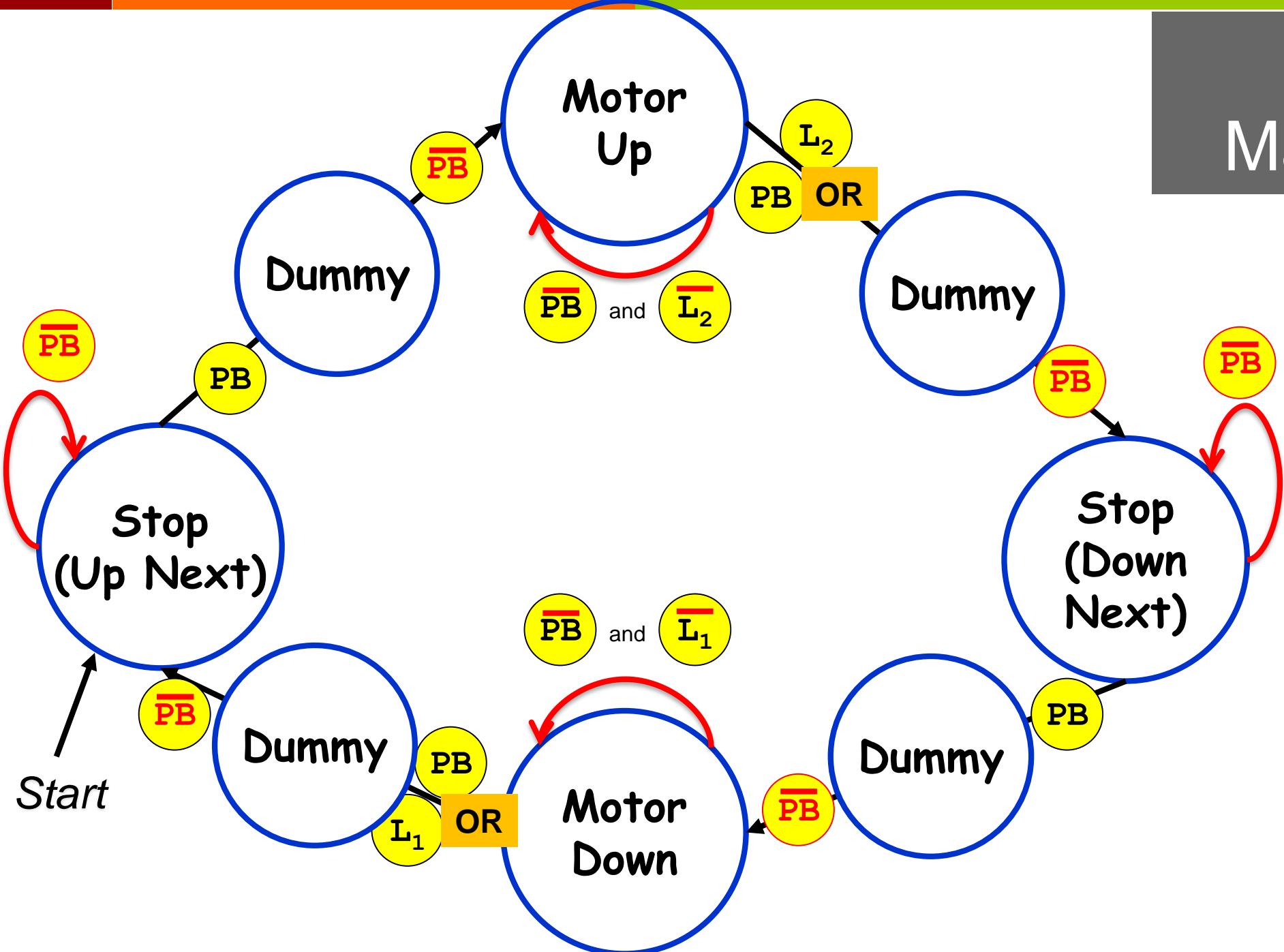


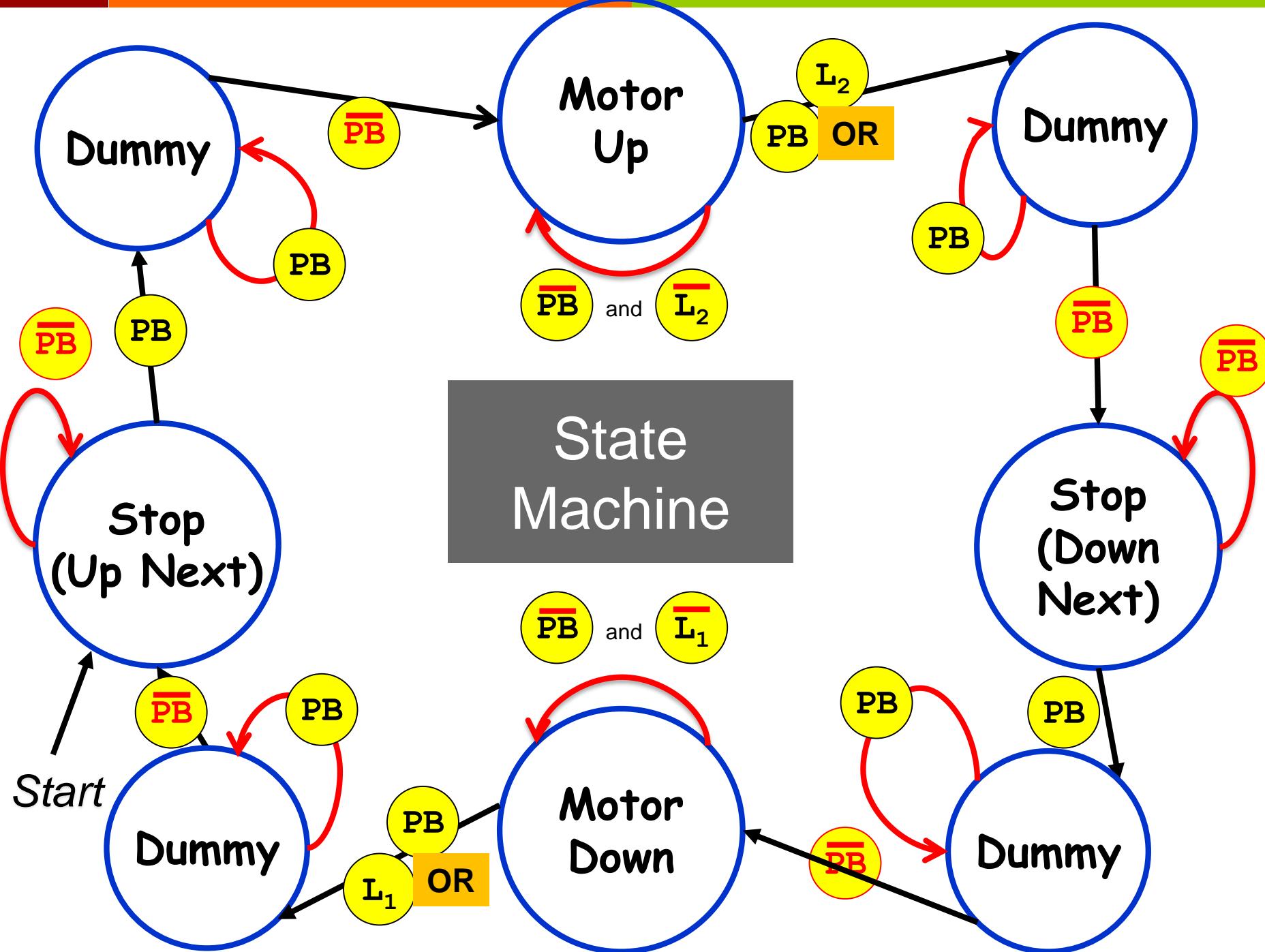
What happens if we just hold down the pushbutton?

State Machine

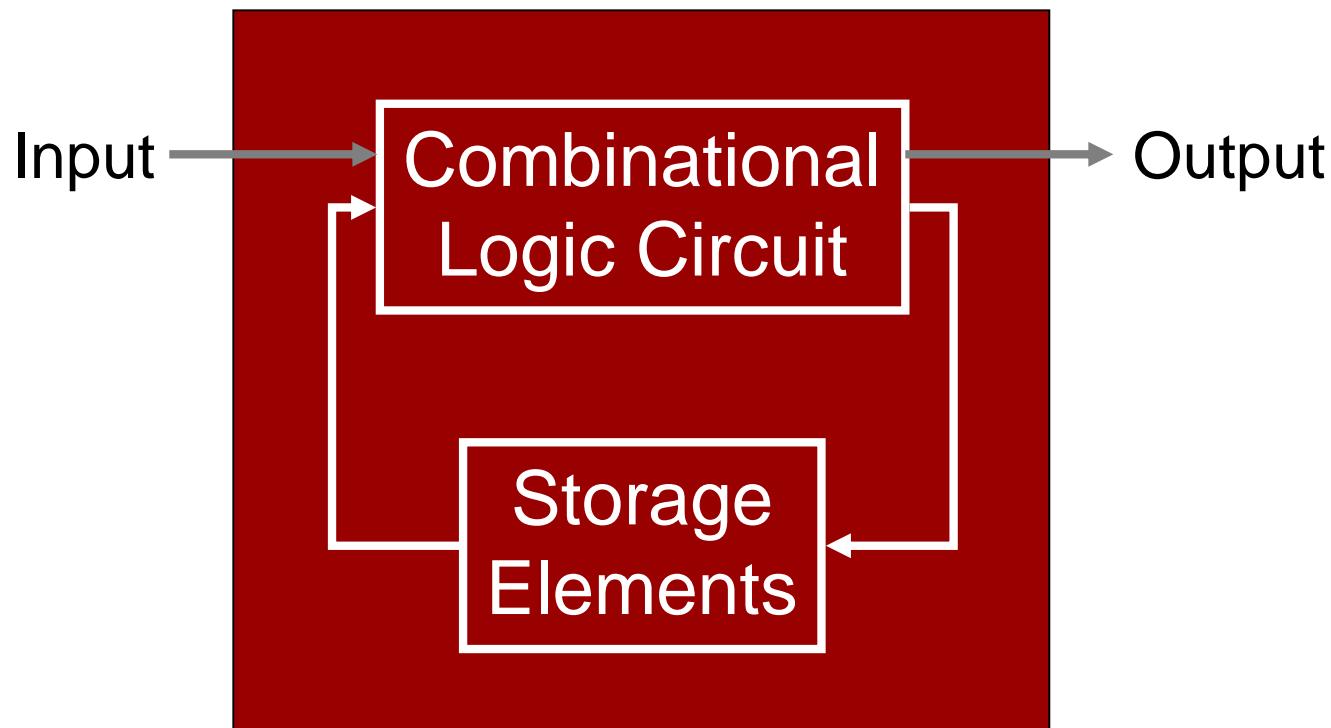


State Machine

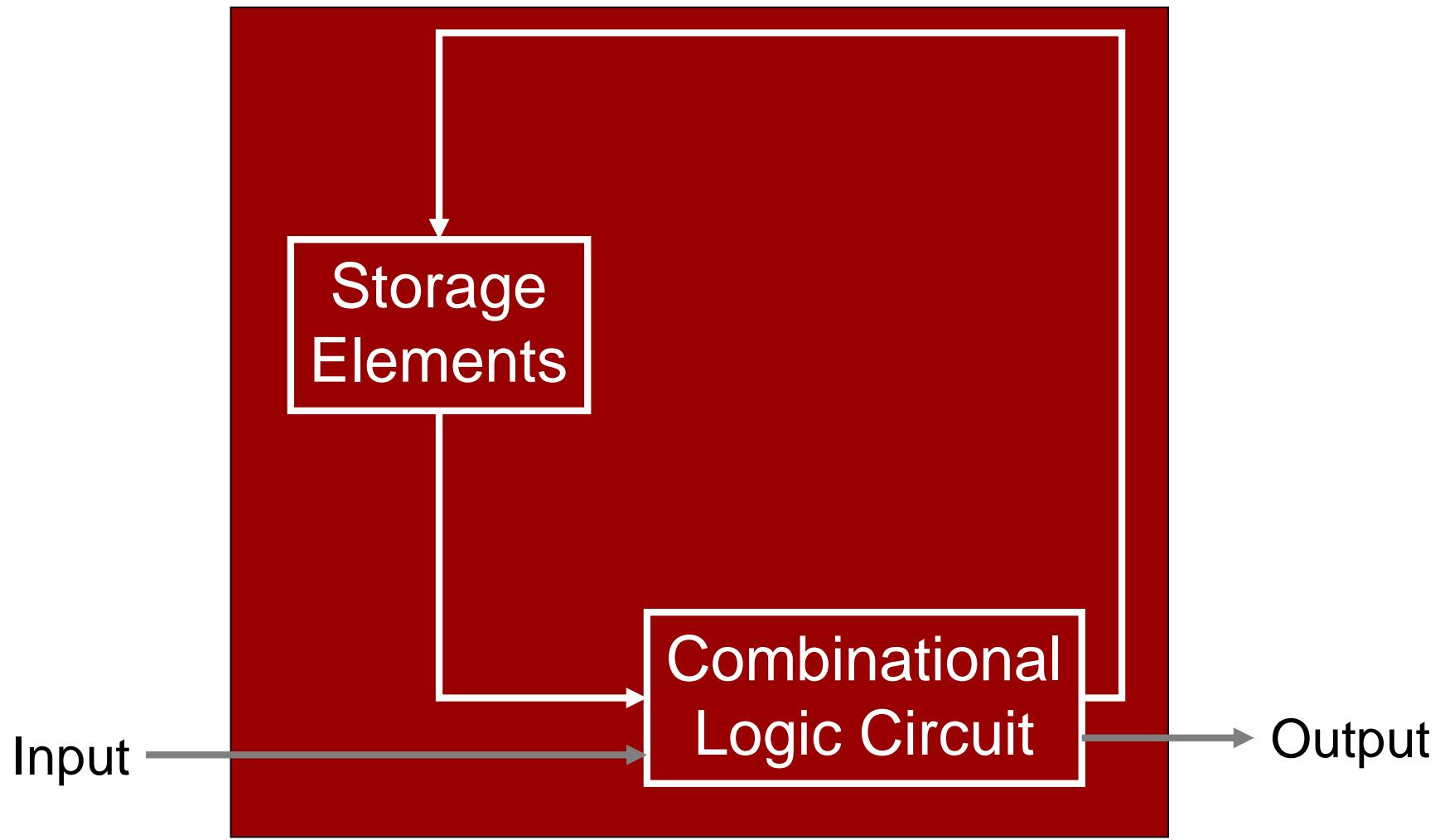




Sequential Logic Circuit



Sequential Logic Circuit

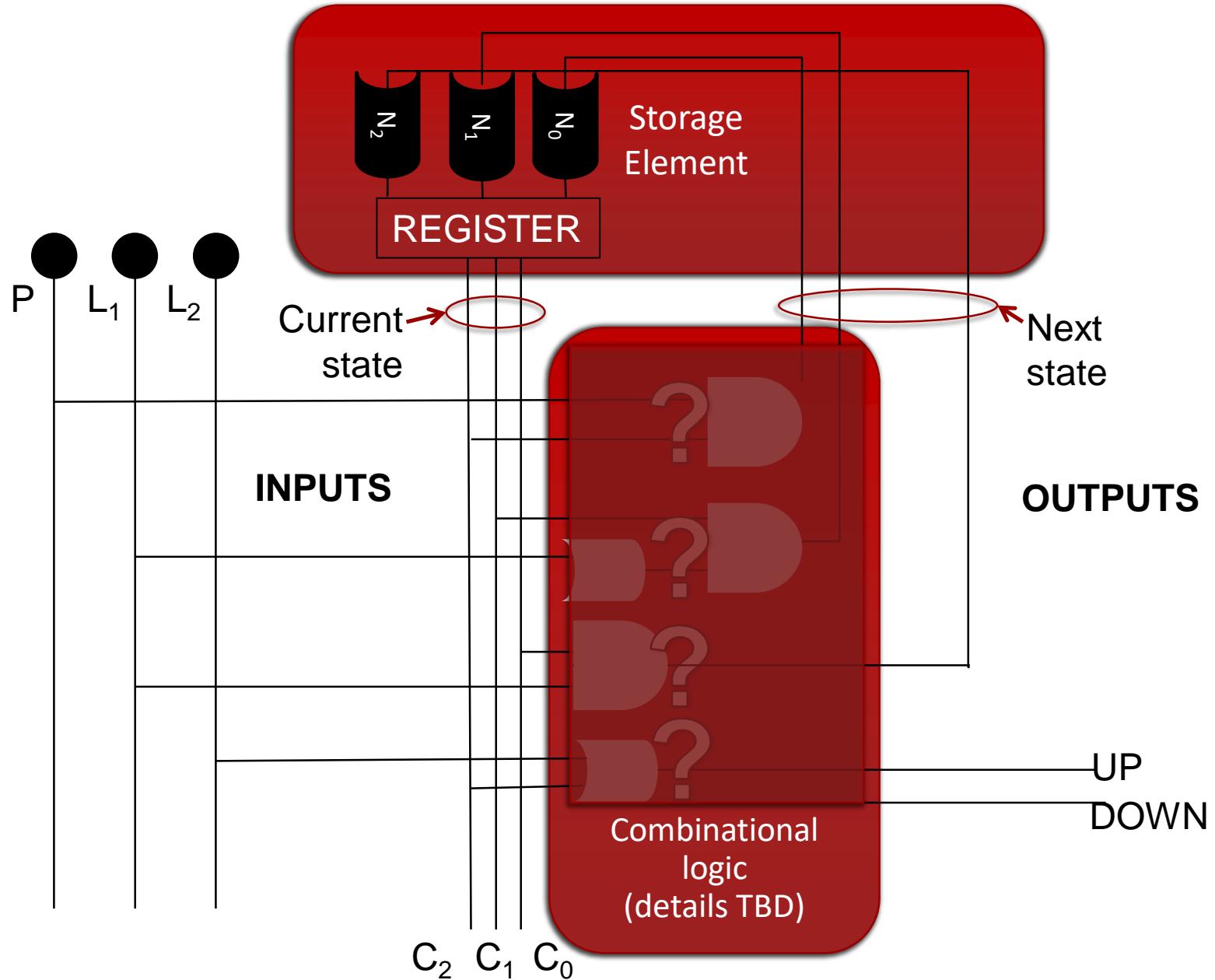


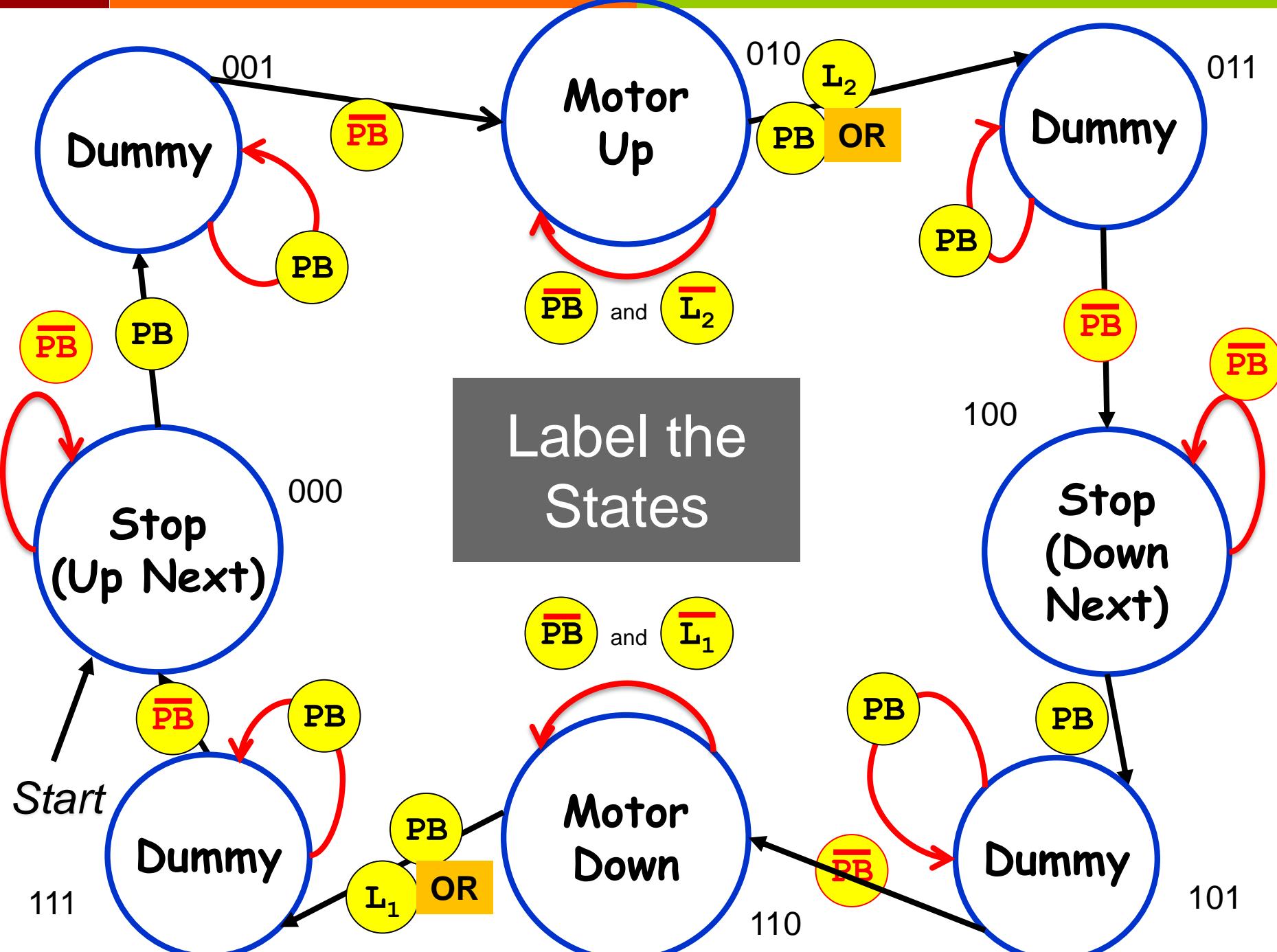
Question

How many bits of storage are we going to need for the garage door opener state machine?

- A. 1
- B. 2
- C. 3 
- D. 4
- E. 5

State Machine Circuit





P	L ₁	L ₂	C ₂	C ₁	C ₀	N ₂	N ₁	N ₀	U	D
0	x	x	0	0	0	0	0	0	0	0
1	x	x	0	0	0	0	0	1	0	0
0	x	x	0	0	1	0	1	0	0	0
1	x	x	0	0	1	0	0	1	0	0
0	x	0	0	1	0	0	1	0	1	0
x	x	1	0	1	0	0	1	1	1	0
1	x	x	0	1	0	0	1	1	1	0
0	x	x	0	1	1	1	0	0	0	0
1	x	x	0	1	1	0	1	1	0	0
0	x	x	1	0	0	1	0	0	0	0
1	x	x	1	0	0	1	0	1	0	0
0	x	x	1	0	1	1	1	0	0	0
1	x	x	1	0	1	1	0	1	0	0

What happens
in state 000

What happens
in state 001

What happens
in state 010

Circuit
Inputs

Current
state

Next state

Circuit
Outputs

P	L_1	L_2	C_2	C_1	C_0	N_2	N_1	N_0	U	D
0	0	x	1	1	0	1	1	0	0	1
1	x	x	1	1	0	1	1	1	0	1
x	1	x	1	1	0	1	1	1	0	1
0	x	x	1	1	1	0	0	0	0	0
1	x	x	1	1	1	1	1	1	0	0



 Circuit Inputs Current state Next state Circuit Outputs

Question

With three inputs and three state bits, how many lines would the truth table require if it didn't contain “don't care” entries?

A. 16

B. 32

C. 64



D. 128

E. 256

Equations from the Truth Table...

$$\begin{aligned} N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\ & + \quad \sim P C_2 \sim C_1 \sim C_0 \\ & + \quad P C_2 \sim C_1 \sim C_0 \\ & + \quad \sim P C_2 \sim C_1 C_0 \\ & + \quad P C_2 \sim C_1 C_0 \\ & + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\ & + \quad P C_2 C_1 \sim C_0 \\ & + \quad L_1 C_2 C_1 \sim C_0 \\ & + \quad P C_2 C_1 C_0 \end{aligned}$$

(Hint: Look down the N_2 column for the 1 bits – these are the input terms that make N_2 true!)

$$AB + \sim AB = B$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad \sim P C_2 \sim C_1 \sim C_0 \\& + \quad P C_2 \sim C_1 \sim C_0 \\& + \quad \sim P C_2 \sim C_1 C_0 \\& + \quad P C_2 \sim C_1 C_0 \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad P C_2 C_1 \sim C_0 \\& + \quad L_1 C_2 C_1 \sim C_0 \\& + \quad P C_2 C_1 C_0\end{aligned}$$

$$AB + \sim AB = B$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad C_2 \sim C_1 \sim C_0 \\& + \quad C_2 \sim C_1 C_0 \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad P C_2 C_1 \sim C_0 \\& + \quad L_1 C_2 C_1 \sim C_0 \\& + \quad P C_2 C_1 C_0\end{aligned}$$

$$A + B = B + A$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + C_2 \sim C_1 \sim C_0 \\& + C_2 \sim C_1 C_0 \\& + \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + P C_2 C_1 \sim C_0 \\& + L_1 C_2 C_1 \sim C_0 \\& + \textcolor{red}{P C_2 C_1 C_0}\end{aligned}$$

$$A + B = B + A$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad P C_2 C_1 C_0 \\& + \quad C_2 \sim C_1 \sim C_0 \\& + \quad C_2 \sim C_1 C_0 \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad P C_2 C_1 \sim C_0 \\& + \quad L_1 C_2 C_1 \sim C_0\end{aligned}$$

$$AC + BC = (A + B)C$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + PC_2 C_1 C_0 \\& + C_2 \sim C_1 \sim C_0 \\& + C_2 \sim C_1 C_0 \\& + \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + PC_2 C_1 \sim C_0 \\& + L_1 C_2 C_1 \sim C_0\end{aligned}$$

$$AB + BC = (A + B)C$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + P C_2 C_1 C_0 \\& + C_2 \sim C_1 \sim C_0 \\& + C_2 \sim C_1 C_0 \\& + \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + (P+L_1) C_2 C_1 \sim C_0\end{aligned}$$

$$AB + \sim AB = B$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad PC_2 C_1 C_0 \\& + \quad C_2 \sim C_1 \sim C_0 \\& + \quad C_2 \sim C_1 C_0 \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad (P+L_1) C_2 C_1 \sim C_0\end{aligned}$$

$$AB + \sim AB = B$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad PC_2 C_1 C_0 \\& + \quad \textcolor{red}{C_2 \sim C_1} \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad (P+L_1) C_2 C_1 \sim C_0\end{aligned}$$

$$AC + BC = (A + B)C$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + \quad PC_2 C_1 C_0 \\& + \quad C_2 \sim C_1 \\& + \quad \sim P \sim L_1 C_2 C_1 \sim C_0 \\& + \quad (P+L_1) C_2 C_1 \sim C_0\end{aligned}$$

$$AC + BC = (A + B)C$$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + P C_2 C_1 C_0 \\& + C_2 \sim C_1 \\& + ((\sim P \sim L_1) + (P + L_1)) C_2 C_1 \sim C_0\end{aligned}$$

DeMorgan and $(A + \sim A)B = B$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + P C_2 C_1 C_0 \\& + C_2 \sim C_1 \\& + (\sim P \sim L_1 + P + L_1) C_2 C_1 \sim C_0\end{aligned}$$

DeMorgan and $(A + \sim A)B = B$

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + P C_2 C_1 C_0 \\& + C_2 \sim C_1 \\& + C_2 C_1 \sim C_0\end{aligned}$$

And We're Down to 4 Variables...

$$\begin{aligned}N_2 = & \quad \sim P \sim C_2 C_1 C_0 \\& + P C_2 C_1 C_0 \\& + C_2 \sim C_1 \\& + C_2 C_1 \sim C_0\end{aligned}$$

$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

How About a K-Map To Go Farther?

$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

	$C_1' C_0'$	$C_1' C_0$	$C_1 C_0$	$C_1 C_0'$
$P'C_2'$				
$P'C_2$				
PC_2				
PC_2'				

Fill Out the Map

$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

	$C_1' C_0'$	$C_1' C_0$	$C_1 C_0$	$C_1 C_0'$
$P'C_2'$			1	
$P'C_2$	1	1		1
PC_2	1	1	1	1
PC_2'				

Look for Power-of-2 Rectangles

$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

	$C_1' C_0'$	$C_1' C_0$	$C_1 C_0$	$C_1 C_0'$
$P'C_2'$			1	
$P'C_2$	1	1		1
PC_2	1	1	1	1
PC_2'				

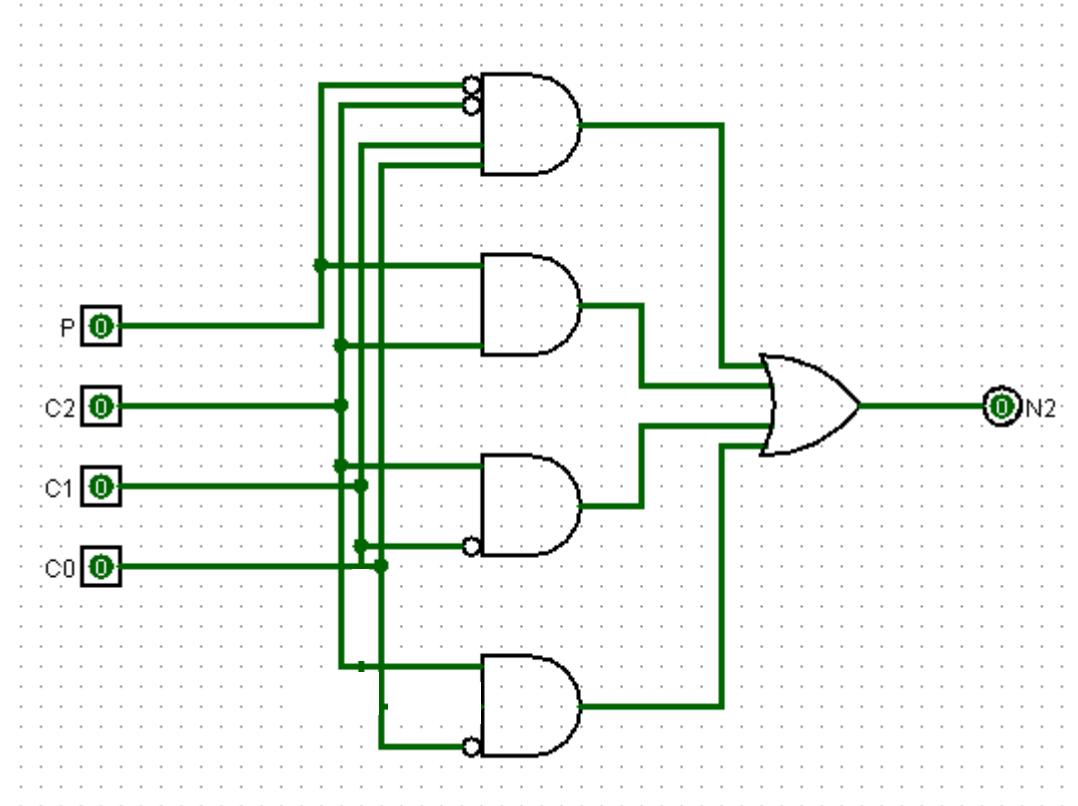
$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

	$C_1' C_0'$	$C_1' C_0$	$C_1 C_0$	$C_1 C_0'$
$P'C_2'$			1	
$P'C_2$	1	1		1
$P C_2$	1	1	1	1
$P C_2'$				

A Karnaugh map for the function N_2 . The columns are labeled $C_1' C_0'$, $C_1' C_0$, $C_1 C_0$, and $C_1 C_0'$. The rows are labeled $P'C_2'$, $P'C_2$, $P C_2$, and $P C_2'$. Red lines highlight the terms $\sim P \sim C_2 C_1 C_0$ and $P C_2 C_1 C_0$. Blue lines highlight the terms $C_2 \sim C_1$ and $C_2 C_1 \sim C_0$.

$$N_2 = \sim P \sim C_2 C_1 C_0 + P C_2 C_1 C_0 + C_2 \sim C_1 + C_2 C_1 \sim C_0$$

Draw the Circuit from the Boolean...



How Many Circuits?

We've designed the circuit for the N2 output. How many more circuits do we need to design to complete the state machine?

A. 1

Why?

B. 2

Go back and look at the truth table

C. 3

We have 5 output columns, each of which needs a circuit to compute it

D. 4



E. 5

Let's look at the circuit for U

P	L ₁	L ₂	C ₂	C ₁	C ₀	N ₂	N ₁	N ₀	U	D
0	x	x	0	0	0	0	0	0	0	0
1	x	x	0	0	0	0	0	1	0	0
0	x	x	0	0	1	0	1	0	0	0
1	x	x	0	0	1	0	0	1	0	0
0	x	0	0	1	0	0	1	0	1	0
x	x	1	0	1	0	0	1	1	1	0
1	x	x	0	1	0	0	1	1	1	0
0	x	x	0	1	1	1	0	0	0	0
1	x	x	0	1	1	0	1	1	0	0
0	x	x	1	0	0	1	0	0	0	0
1	x	x	1	0	0	1	0	1	0	0
0	x	x	1	0	1	1	1	0	0	0
1	x	x	1	0	1	1	0	1	0	0

{Circuit Inputs} {Current state} {Next state} {Circuit Outputs}

The only place
U=1 is here

P	L_1	L_2	C_2	C_1	C_0	N_2	N_1	N_0	U	D
0	0	x	1	1	0	1	1	0	0	1
1	x	x	1	1	0	1	1	1	0	1
x	1	x	1	1	0	1	1	1	0	1
0	x	x	1	1	1	0	0	0	0	0
1	x	x	1	1	1	1	1	1	0	0



 Circuit Inputs Current state Next state Circuit Outputs

One More Example

$$\Rightarrow U = P'L_2'C_2'C_1C_0' \\ + L_2C_2'C_1C_0' \\ + PC_2'C_1C_0'$$

$$A + A'B = A + B$$

↗ $U = P'L_2'C_2'C_1C_0'$
 + $L_2C_2'C_1C_0'$
 + $PC_2'C_1C_0'$

$$A + A' = 1$$

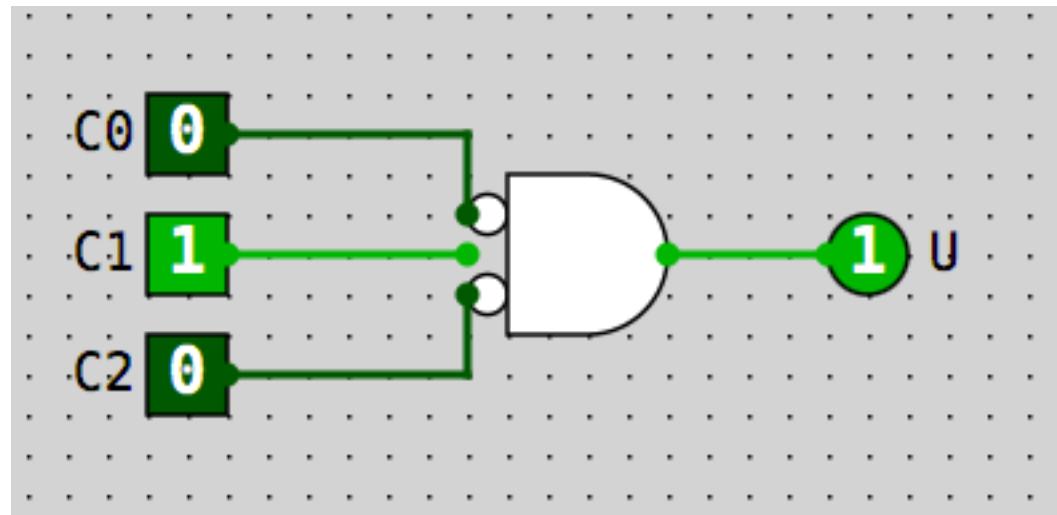
↗ $U = P'C_2'C_1C_0'$
+ $L_2C_2'C_1C_0'$
+ $PC_2'C_1C_0'$

$$A + AB = A$$

↗ $U = C_2' C_1 C_0'$
+ $L_2 C_2' C_1 C_0'$

At last!

↗ $U = C_2' C_1 C_0'$



When is the only time U is 1?

And we must still make circuits for N_1 , N_0 , and D

The Other Three Circuits

↗ $N_1 = C_1 \sim C_0 + PC_1 + \sim P \sim C_1 C_0$

↗ $N_0 = P$

↗ $D = C_2 C_1 \sim C_0$

Two Kinds of State Machines!

One Hot

- ↗ One bit per state
- ↗ Only one bit is on at a time
- ↗ Faster
- ↗ Requires more flip flops
- ↗ States progress
00001»00010»00100»
01000»10000

Binary Encoded

- ↗ Encode state as a binary number
- ↗ Use a decoder to generate a line for each state
- ↗ Slower
- ↗ More complicated
- ↗ States progress
000»001»010»011»100

Which one did we just use to implement the garage door opener?

- ↗ Sequential Logic Circuits
 - ↗ State
 - ↗ Memory
 - ↗ Address space
 - ↗ Addressability
 - ↗ $2^2 \times 3$ bit memory
 - ↗ Clocks
 - ↗ Edge-triggered and level-triggered flip-flops
 - ↗ Example State Machine
 - ↗ Design
 - ↗ Simplification
 - ↗ Implementation
 - ↗ One-hot and Encoded State Machines

LC-3 Datapath



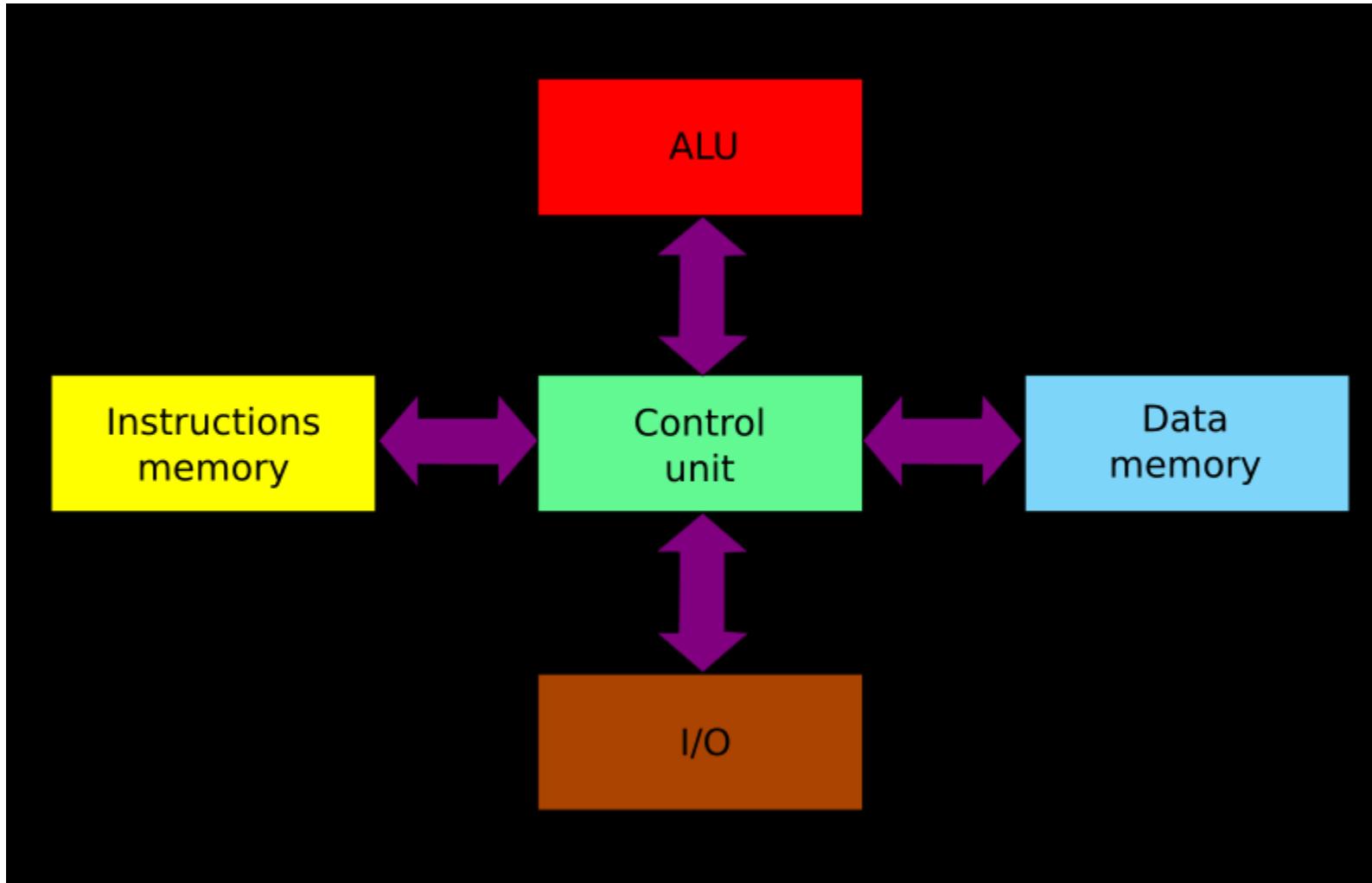
Chapter 4

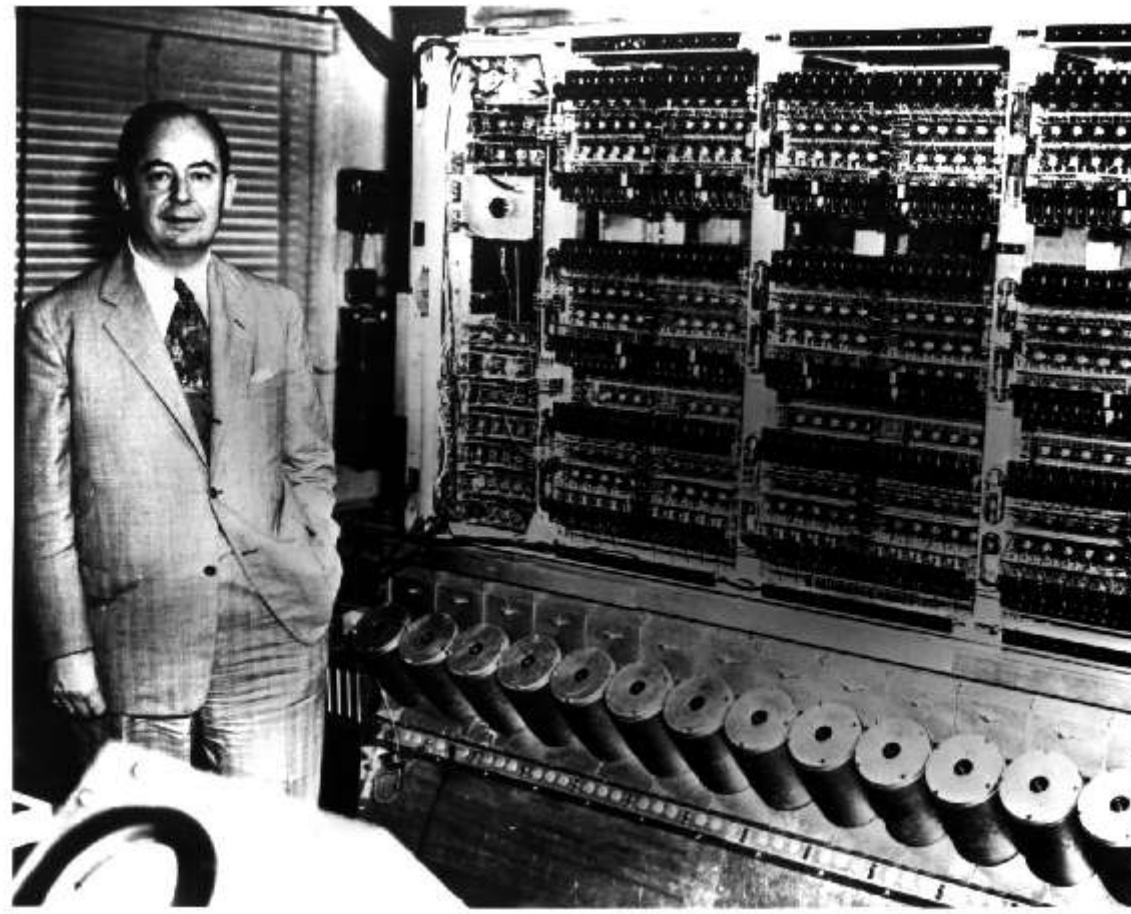
- ↗ Basic Components
 - ↗ Memory, Processing Unit, Input & Output, Control Unit
- ↗ Architectures
 - ↗ Harvard
 - ↗ Von Neumann
- ↗ LC-3: A von Neumann Architecture
- ↗ Tri-State Buffers
- ↗ Instruction Cycle
 - ↗ Instructions
 - ↗ Fetch, Decode, Evaluate Address, Fetch Operands, Execute, Store Result



Harvard Mark I

Harvard Model



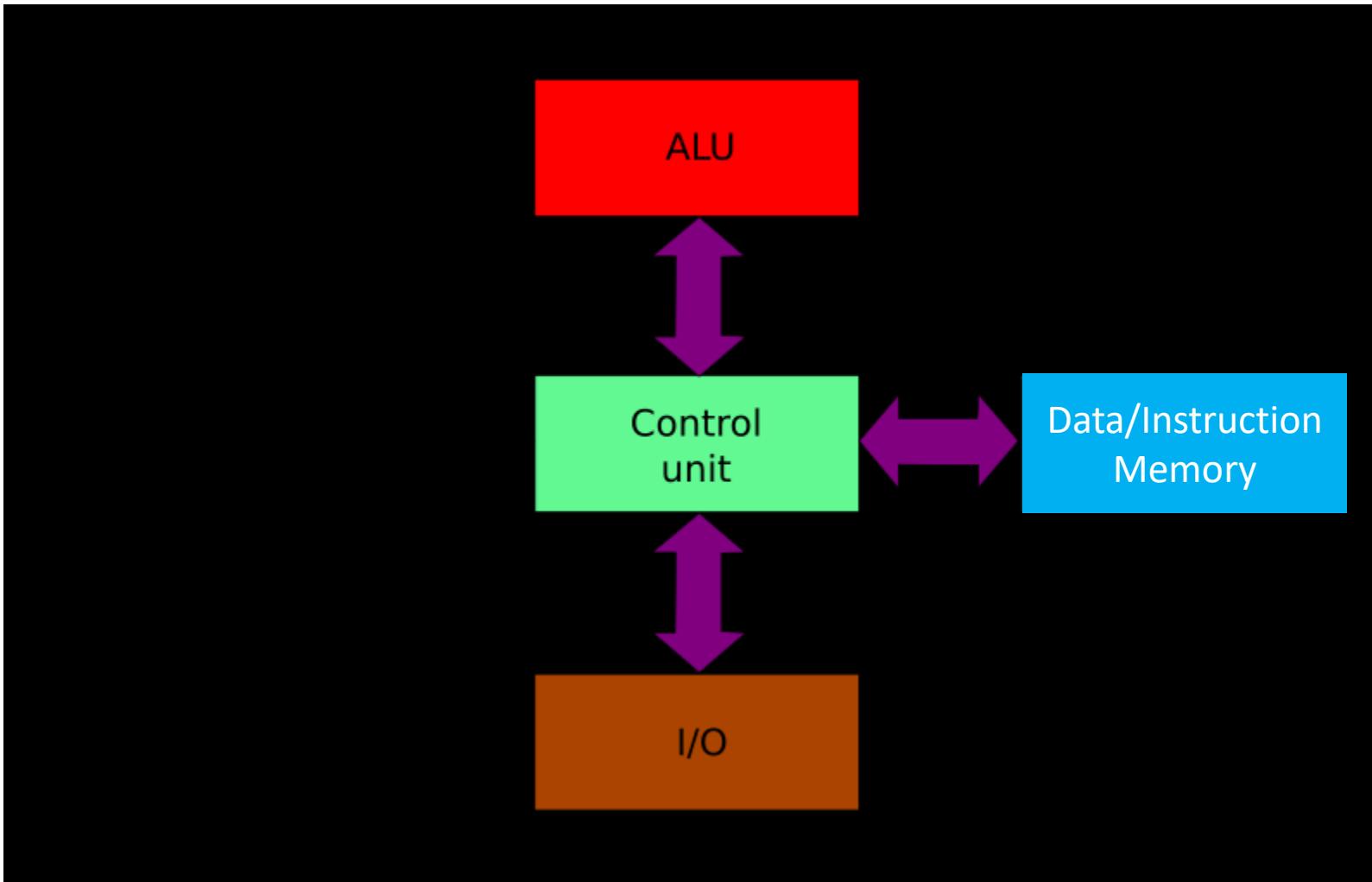


John von Neumann & EDVAC

Electronic Discrete Variable Automatic Computer

First Draft of a Report on the EDVAC
by John von Neumann,
Contract No. W-670-ORD-4926,
Between the United States Army Ordnance Department
and the University of Pennsylvania Moore School of
Electrical Engineering
University of Pennsylvania
June 30, 1945

Von Neumann Model



- The central idea in the von Neumann model of computer processing is that the ***program*** and ***data*** are both stored as sequences of bits in the ***computer's memory***, and the program is executed, one instruction at a time, under the direction of the control unit.

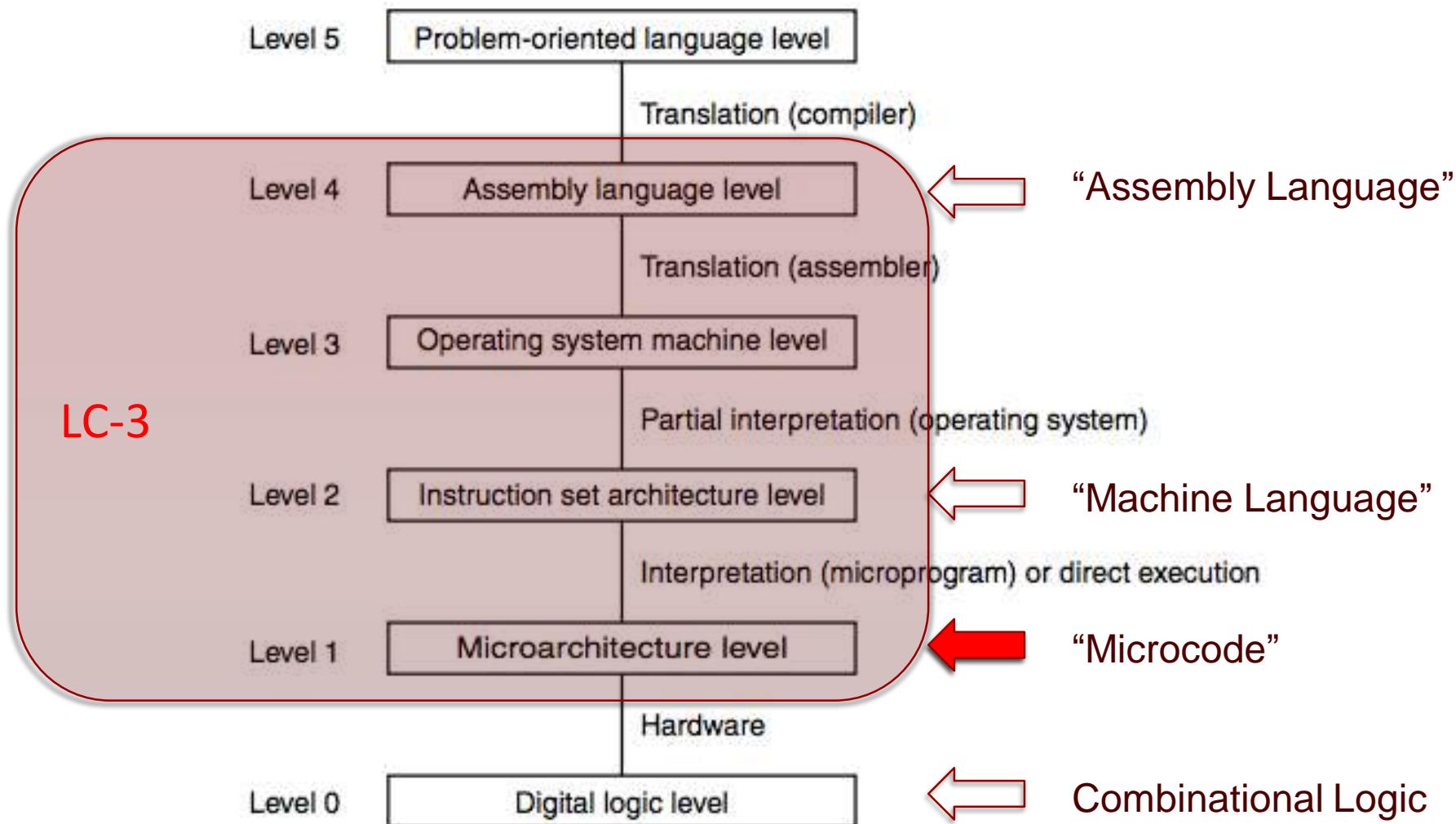
Machine Instructions as Data?

- ↗ The von Neumann model leads us to treat machine instructions as just another data representation!
 - ↗ *This is a big deal*
- ↗ Think of
 - ↗ unsigned integer
 - ↗ twos-complement integer
 - ↗ ASCII
 - ↗ IEEE-754 floating point
 - ↗ ...
 - ↗ **machine instructions**
- ↗ They are all data representations
 - ↗ *(Bits are just bits!)*

- ↗ What can be stored in memory?
 - ↗ Data
 - ↗ Instructions

- ↗ How many memories do we need?
 - ↗ Just one, according to von Neumann

Where are we in this course?

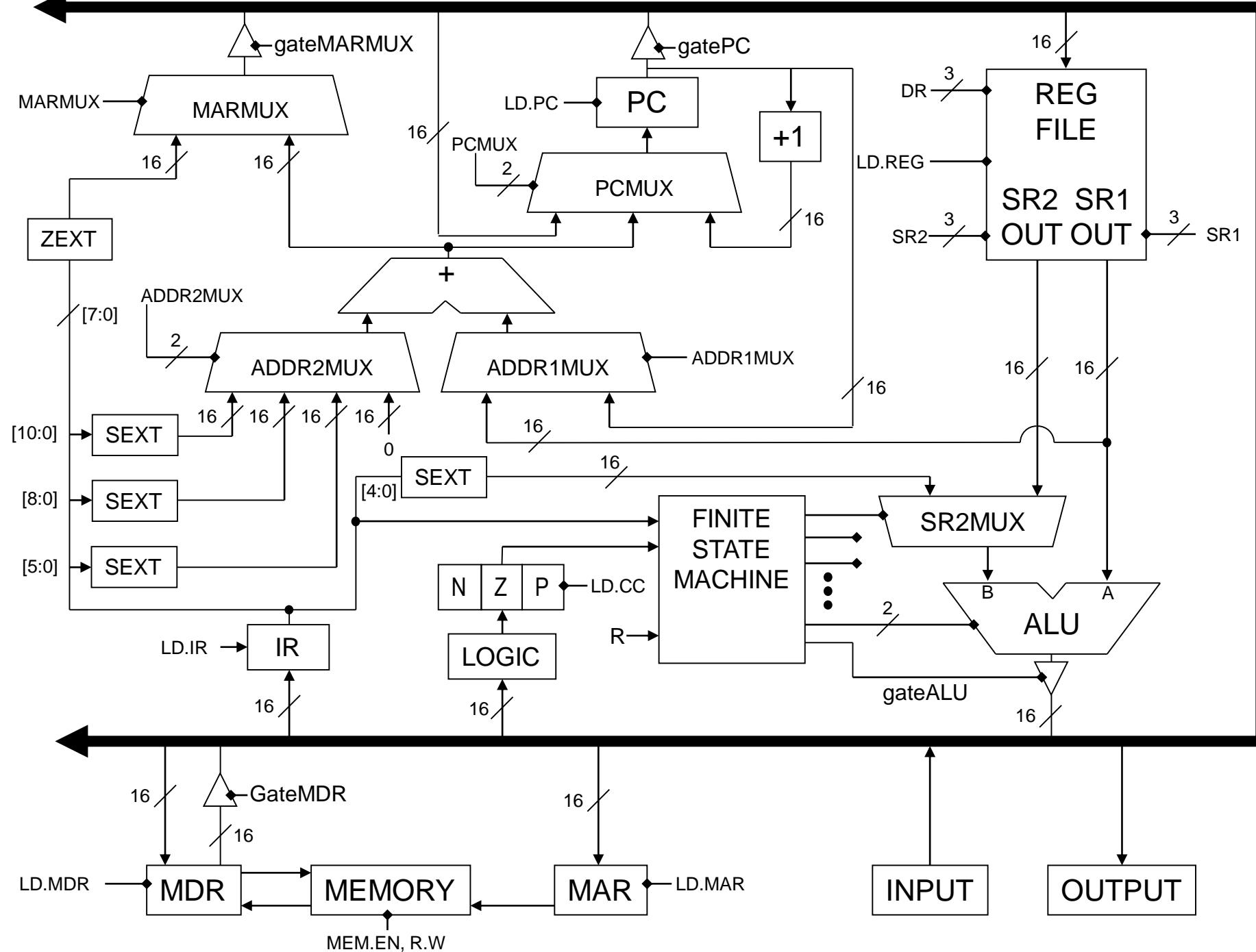


Introducing the LC-3 CPU!

- Address space
 - $2^{16} = 65536$
- Addressability
 - 16 bits
- Architecture type
 - Von Neumann
- General purpose registers
 - 8
- Instruction size
 - 16 bits

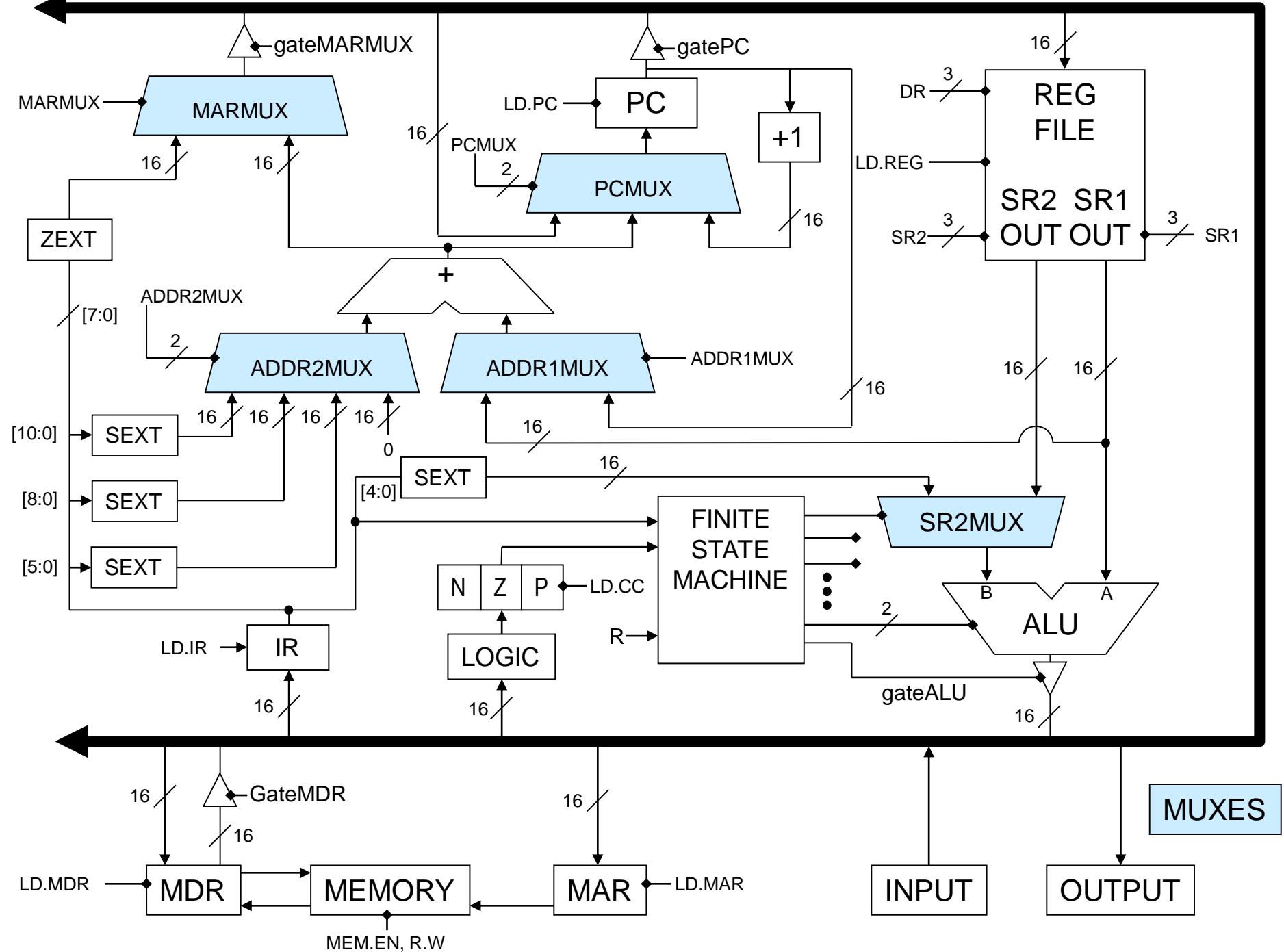
The LC-3 Datapath

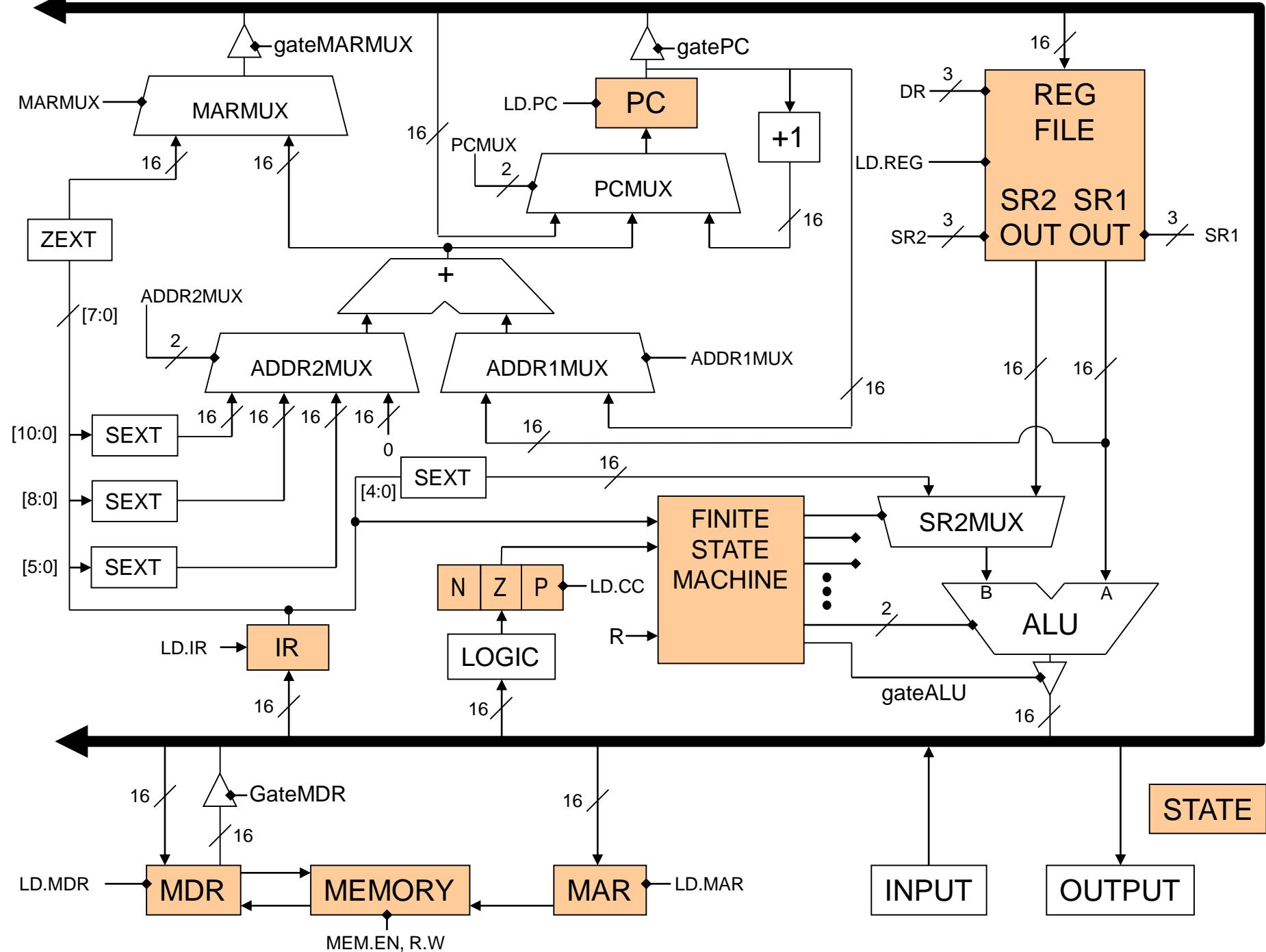
- ↗ Let's build a CPU
 - ↗ Using digital logic – that we already know!

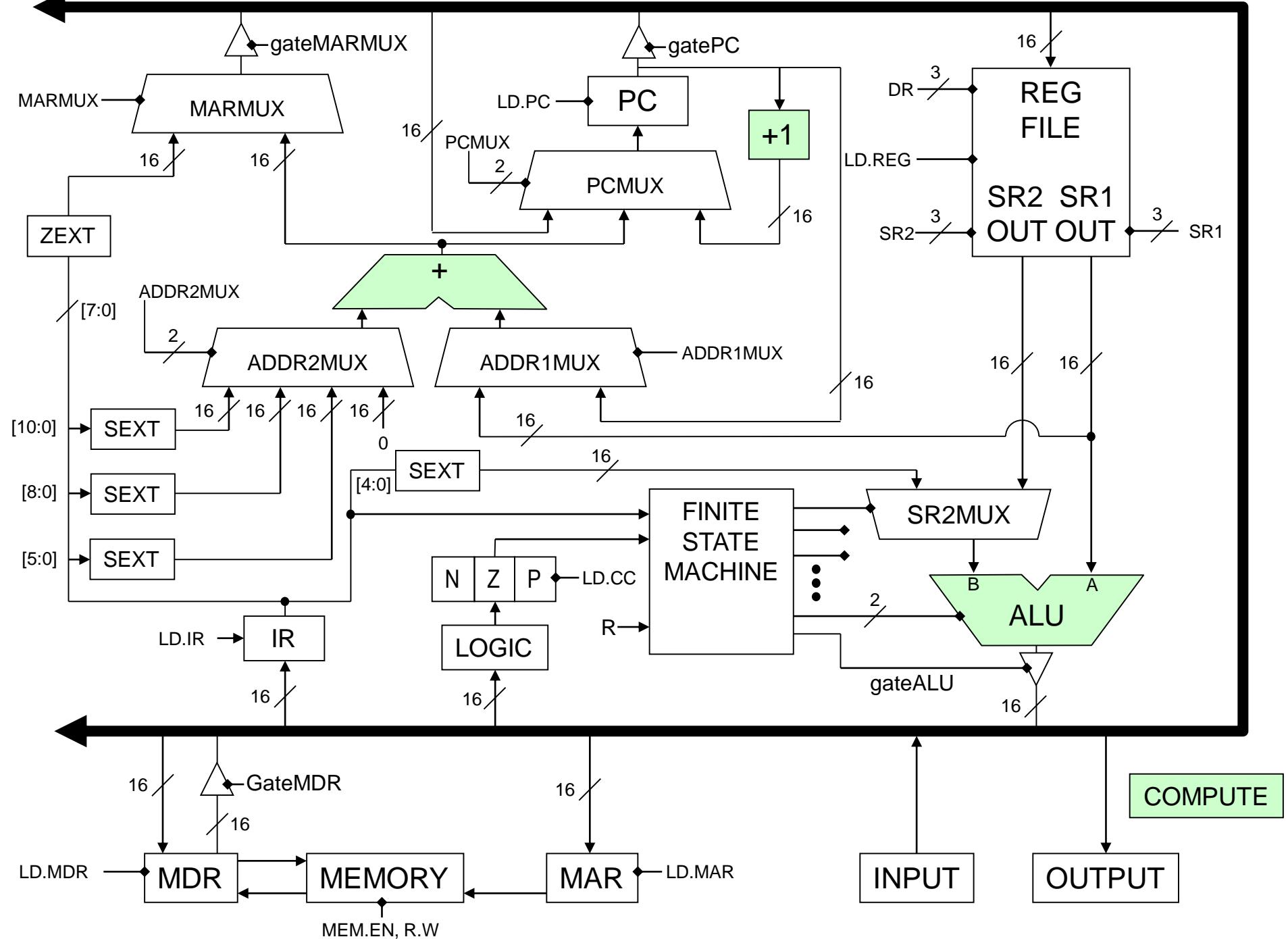


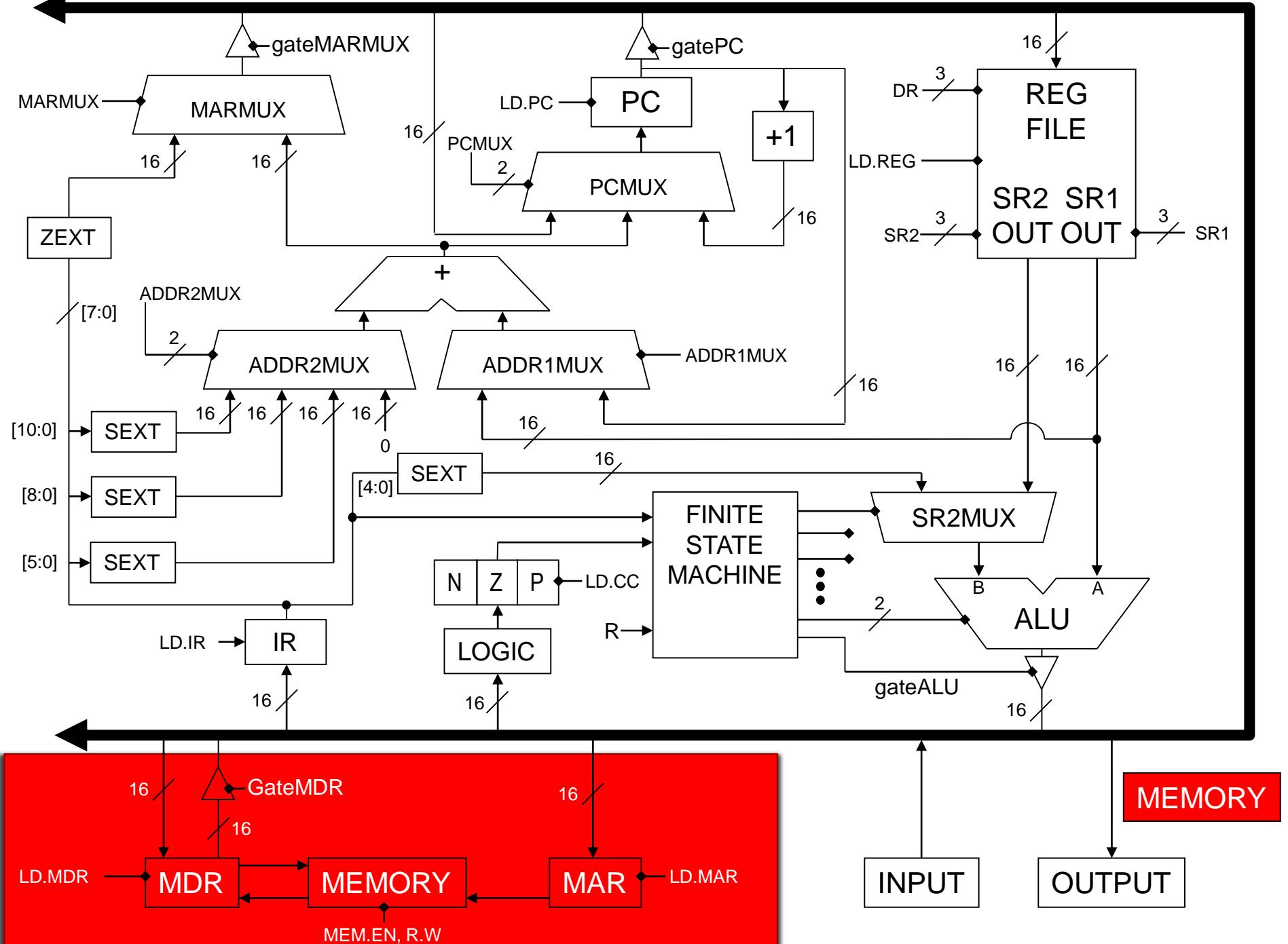
**LC-3
Datapath**

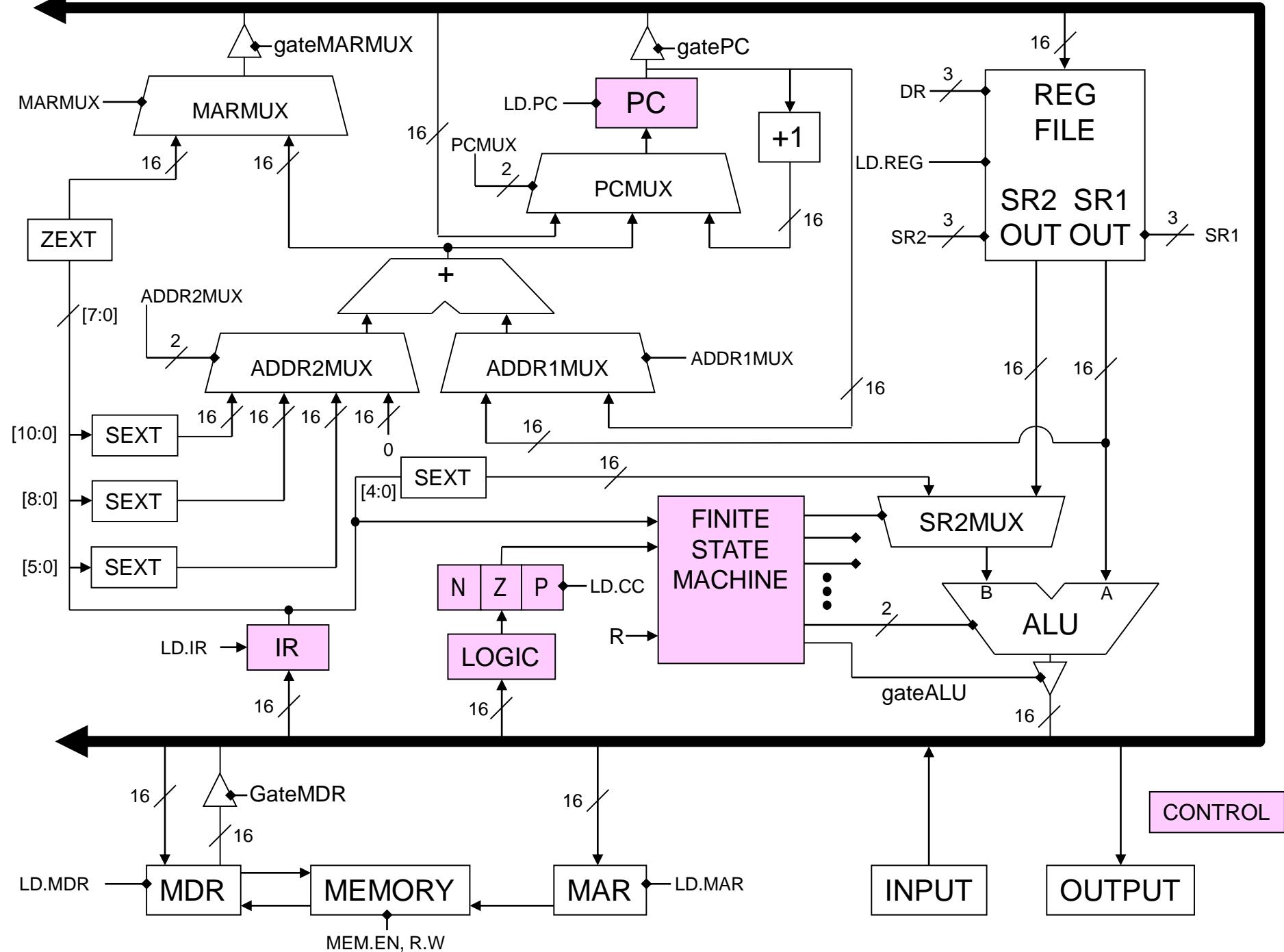
- ↗ That looks scary!
- ↗ But you already know what nearly all of these components are.
- ↗ Let's break it down, piece by piece.





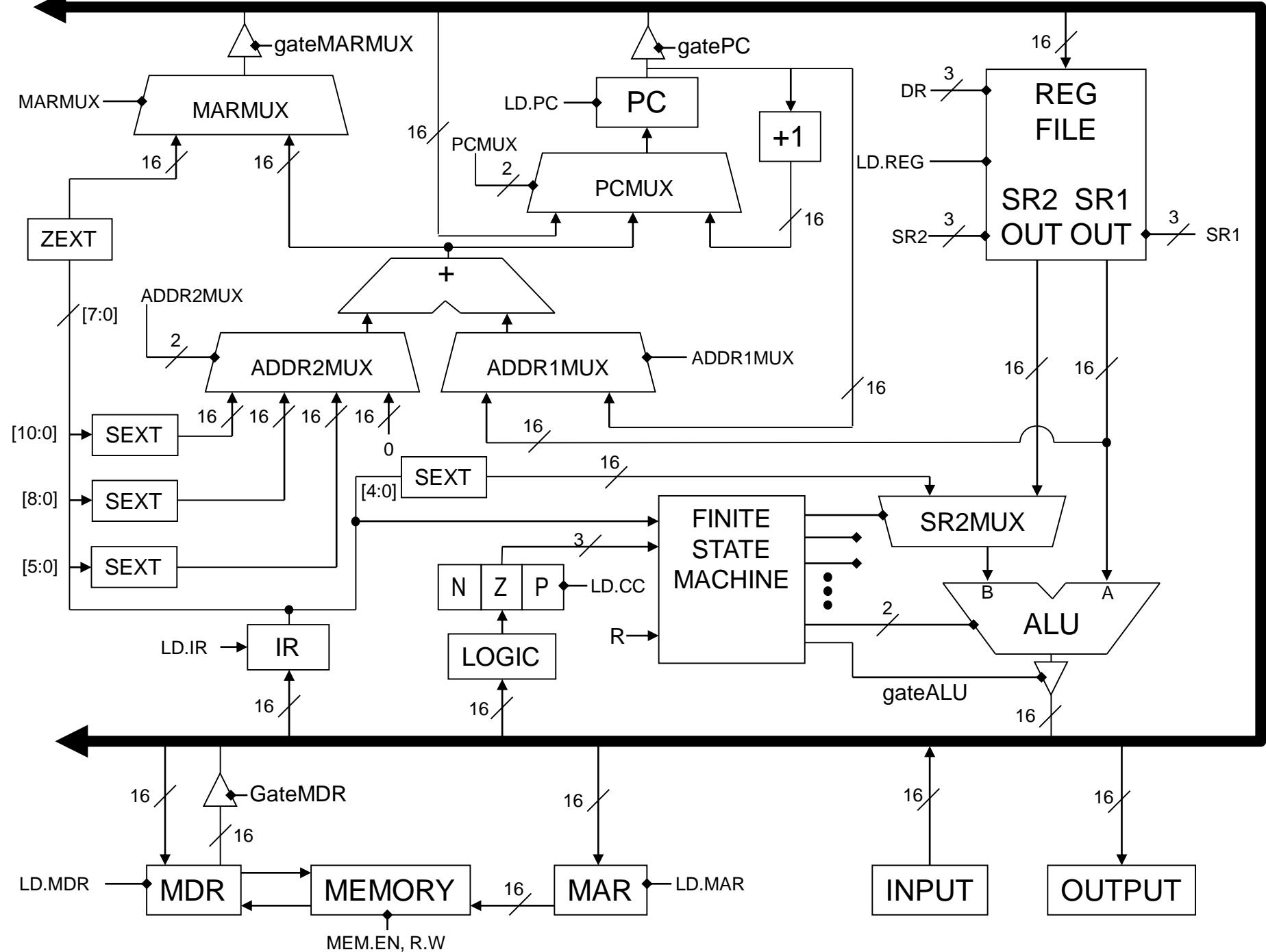






One more Digital Logic Component ...

- ↗ There is one more combinational logic circuit we have not yet introduced:
 - ↗ **Tri-State Buffer**
- ↗ The tri-state buffers are used to help with the **bus**
 - ↗ The bus is 16 shared wires in the datapath
 - ↗ (No transistors or logic, just wires to connect things)
 - ↗ This is what is used to disconnect inputs so the bus can be shared among multiple components
- ↗ Patt names all his tri-state buffers with the prefix “gate”



Important Concept Approaching

- ↗ Remember – our logic gates literally connect their output to $+V_{CC}$ for 1 and Ground for 0
- ↗ When sharing a wire with another circuit – this arrangement is called a **bus**
- ↗ What happens when you connect a 1 ($+V_{CC}$) to a 0 (ground)?
- ↗ Hint: It involves converting electrical energy into heat energy. A lot of it.

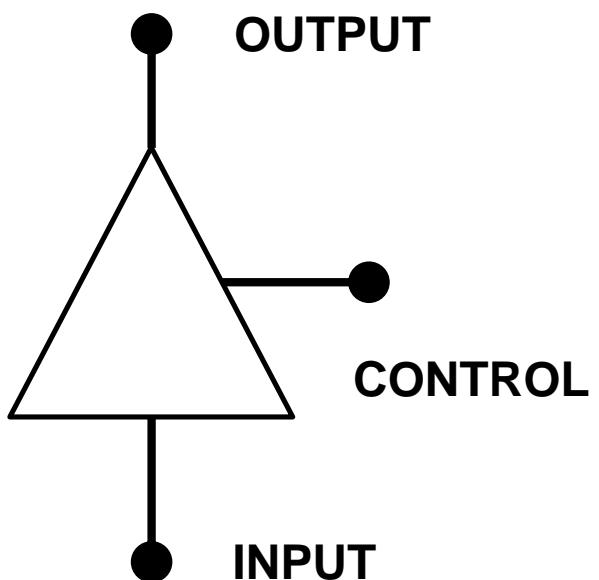
Melted Wires, Smoke, Flames...



ONLY YOU

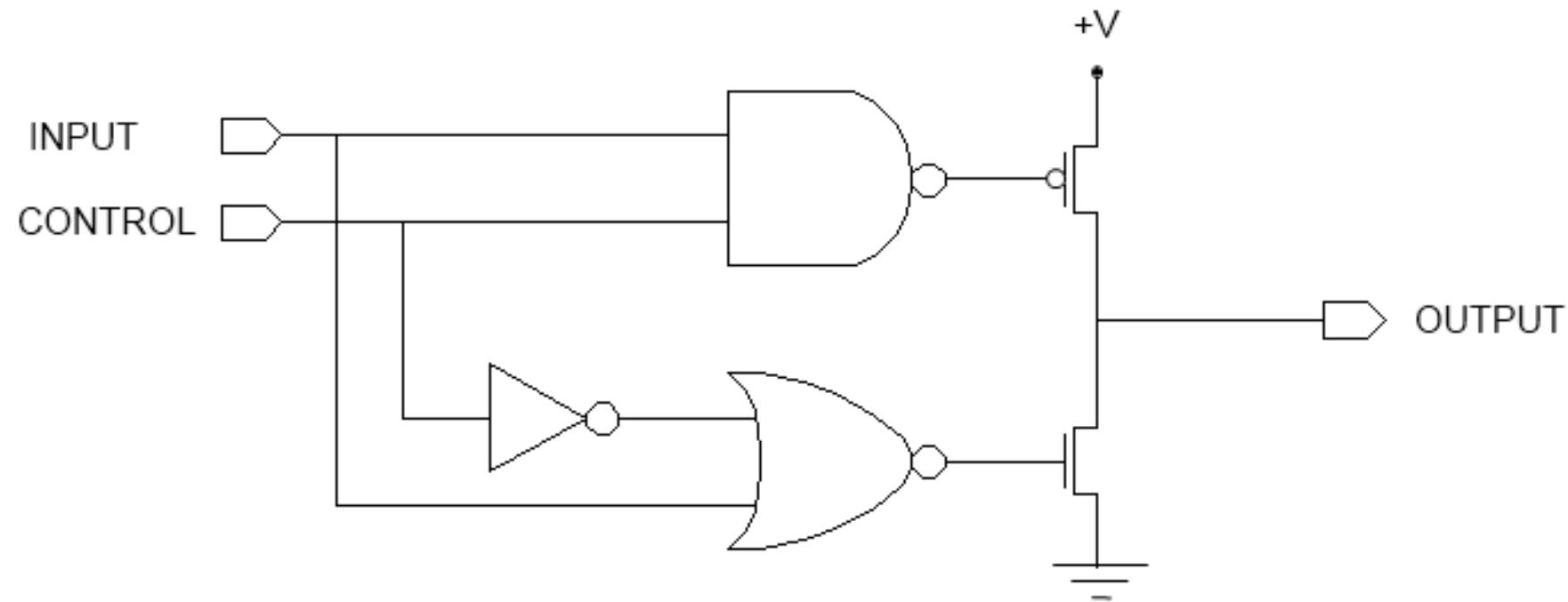
**Can Prevent Datapath Fires!
Only assert one Gate signal
per clock cycle!**

Tri State Buffer



CONTROL	INPUT	OUTPUT
0	0	Z
0	1	Z
1	0	0
1	1	1

Tri State Buffer



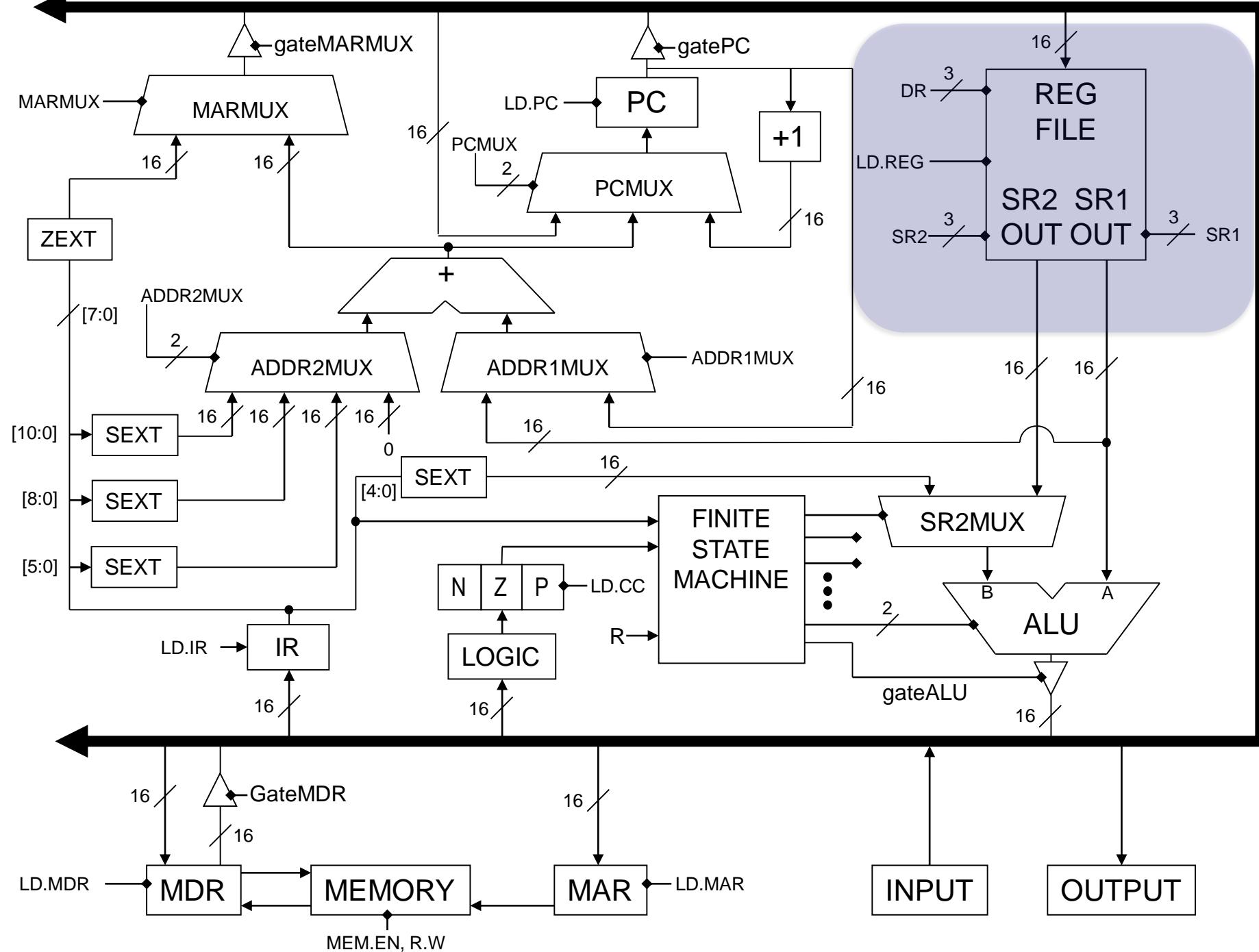
Tri-state buffers are used to

- A. Prevent data path fires
- B. Avoid short circuits
- C. Disconnect a circuit from a bus so that another circuit can assert a value on the same bus without interference
- D. Present a value that is neither 1 or 0 on an output
- E. All of the above



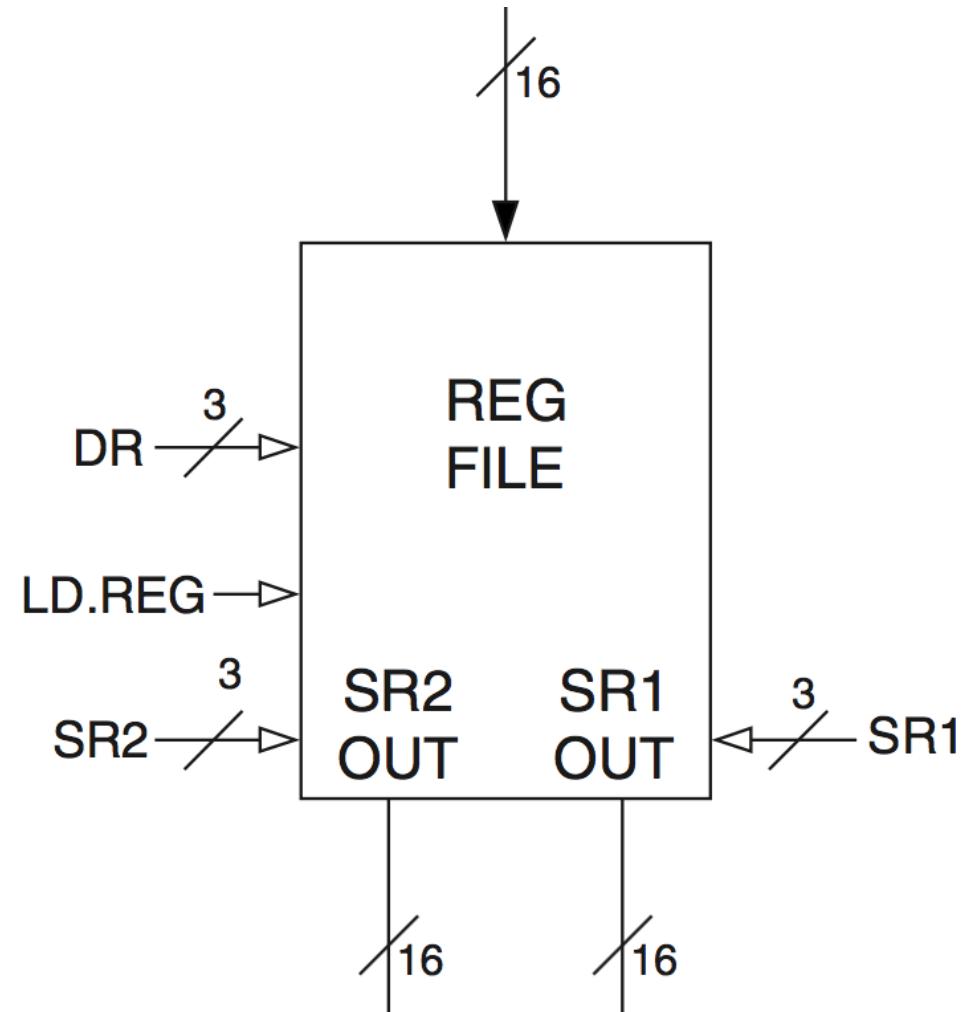
Questions?

Register File



Register File Circuit

- A small, fast “Memory”
 - Address Space: 8 registers
 - 16-bit addressability per reg
- Two outputs
 - Dual-ported memory
 - Can read two regs at same time
 - SR1 (3 bit address)
 - SR2 (3 bit address)
- One input
 - DR (3 bit address)
 - LD.REG (write enable)
- Can read two registers and write one register in a single clock cycle(!)



What is a Machine Instruction?

- ↗ It's just another data representation!
- ↗ It tells the processor what to do next
- ↗ Is it a datatype?
- ↗ Yes. There is definitely hardware to interpret it.

Categories of Machine Instructions

Operate (ALU)	Data Movement (Memory)	Control				
<ul style="list-style-type: none">• ADD• AND• NOT	<table><thead><tr><th><u>Load</u></th><th><u>Store</u></th></tr></thead><tbody><tr><td><ul style="list-style-type: none">• LD• LDR• LDI• LEA</td><td><ul style="list-style-type: none">• ST• STR• STI</td></tr></tbody></table>	<u>Load</u>	<u>Store</u>	<ul style="list-style-type: none">• LD• LDR• LDI• LEA	<ul style="list-style-type: none">• ST• STR• STI	<ul style="list-style-type: none">• BR• JMP• JSR• JSRR• RET• RTI• TRAP
<u>Load</u>	<u>Store</u>					
<ul style="list-style-type: none">• LD• LDR• LDI• LEA	<ul style="list-style-type: none">• ST• STR• STI					

We'll explain what each of these instructions does

- Just briefly enough for now, so you know how to make them happen in the datapath
- With control signals, that come from the state machine

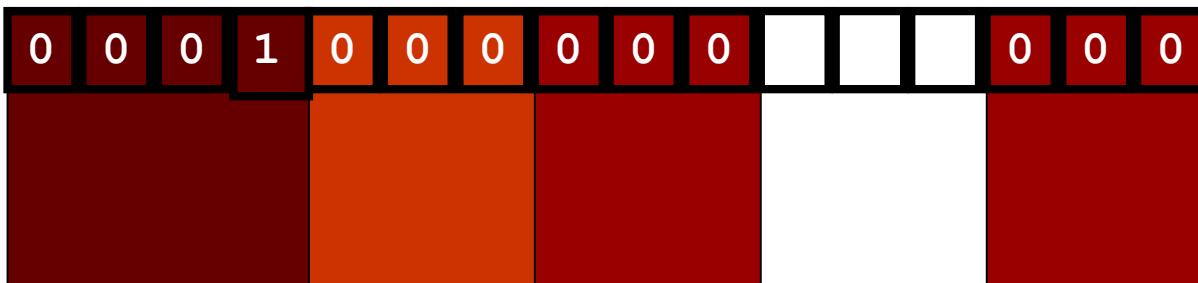
Some LC-3 Instructions

ADD ⁺	0001	DR	SR1	0	00	SR2
ADD ⁺	0001	DR	SR1	1		imm5
AND ⁺	0101	DR	SR1	0	00	SR2
AND ⁺	0101	DR	SR1	1		imm5
BR	0000	n	z	p		PCoffset9
JMP	1100	000	BaseR		000000	
JSR	0100	1			PCoffset11	
JSRR	0100	0	00	BaseR		000000
LD ⁺	0010	DR			PCoffset9	
LDI ⁺	1010	DR			PCoffset9	

⁺ Indicates instructions that modify condition codes

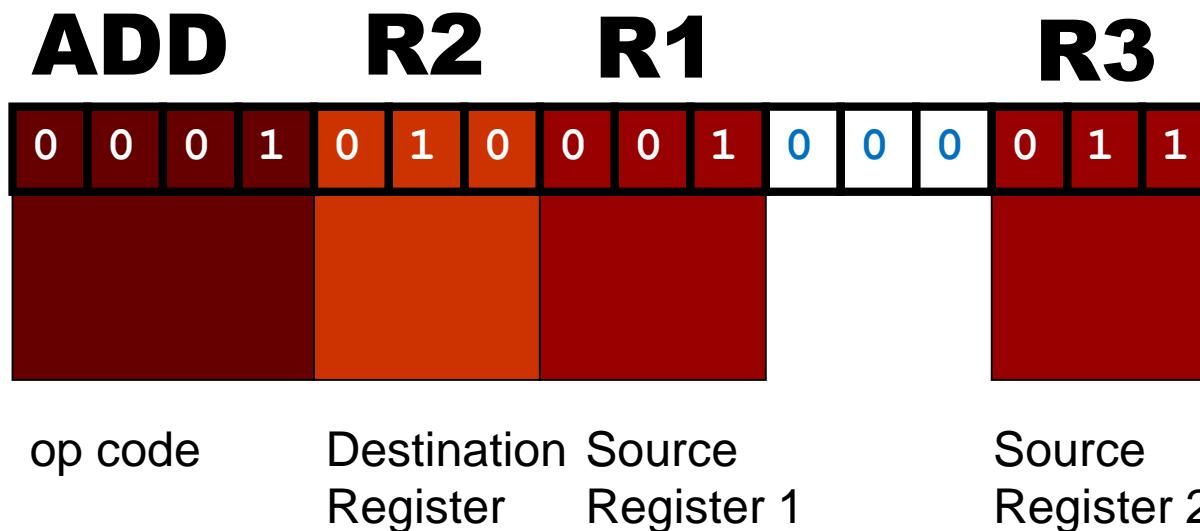
LC-3 Instructions

ADD



Instruction Example

ADD R2,R1,R3



0x1443
012103

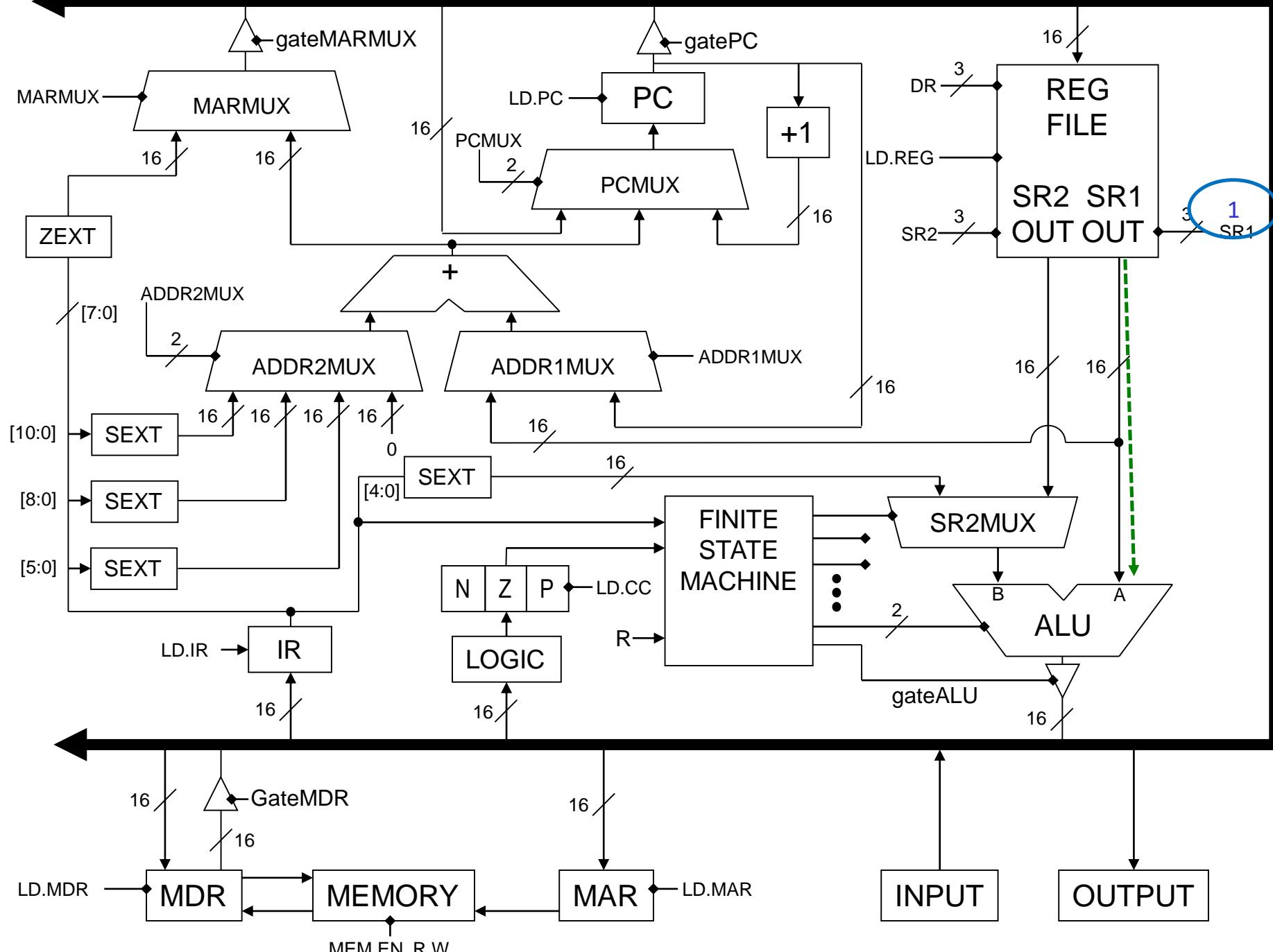
This instruction means:
 $R2 = R1 + R3$

How do we make the LC-3 do an “ADD”?

- The LC-3 is controlled by a large finite state machine.
- So it has output signals (**control signals**)
 - Like UP and DOWN signals in the garage door
- The FSM turns on specific control signals in the LC-3
 - This activates certain paths in the datapath.
 - These control signals can make the datapath do things, such as add two numbers (R1 and R3), and store the result in R2.
- Example:
 - ADD R2, R1, R3
 - This means, $R2 = R1 + R3$
 - *Assume the data is already in R1 and R3 in the register file*

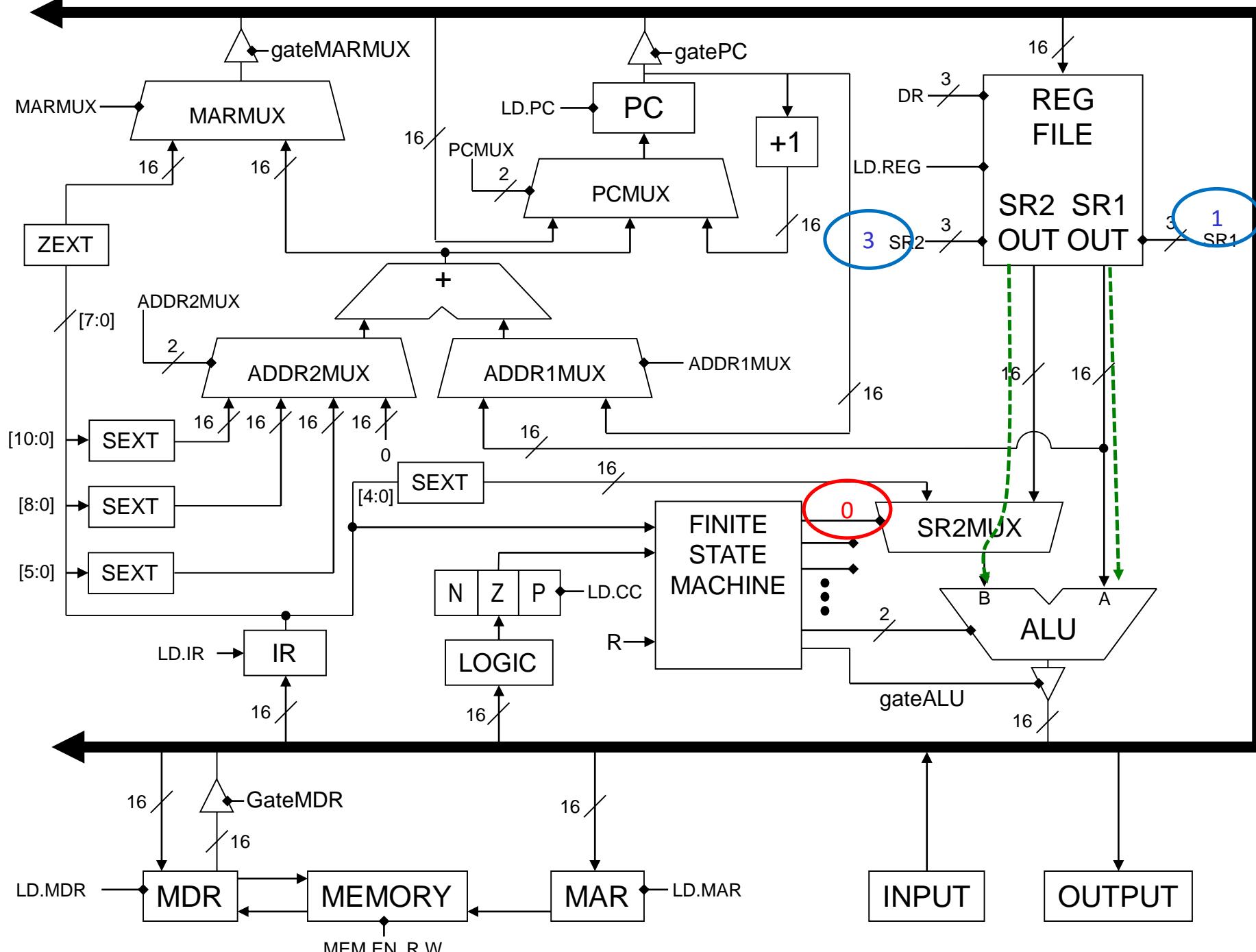
ADD R2, R1, R3 (Execute)

- Copy R1 to ALU
Input A



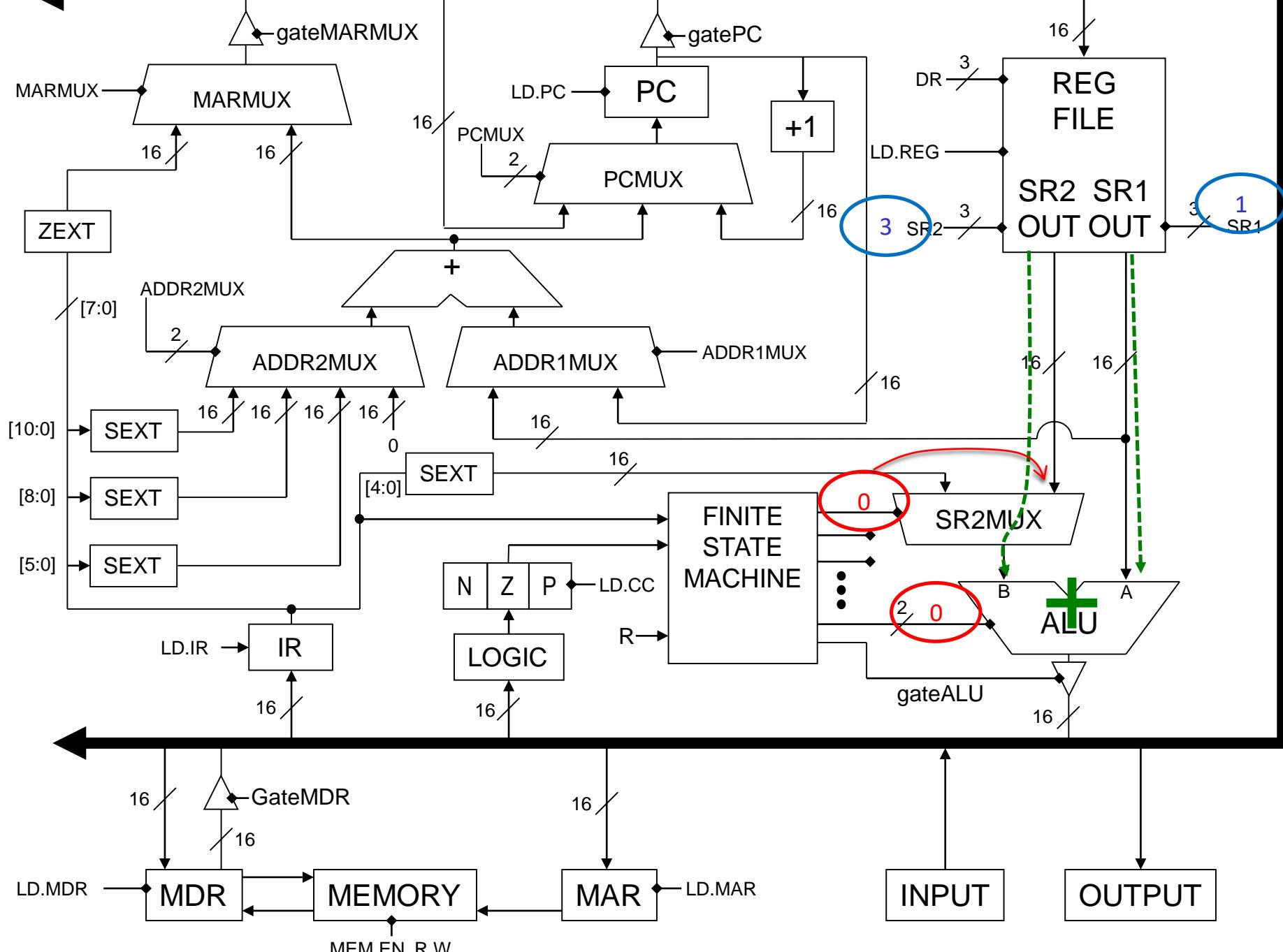
ADD R2, R1, R3 (Execute)

- Copy R1 to ALU Input A
- Copy R3 to ALU input B



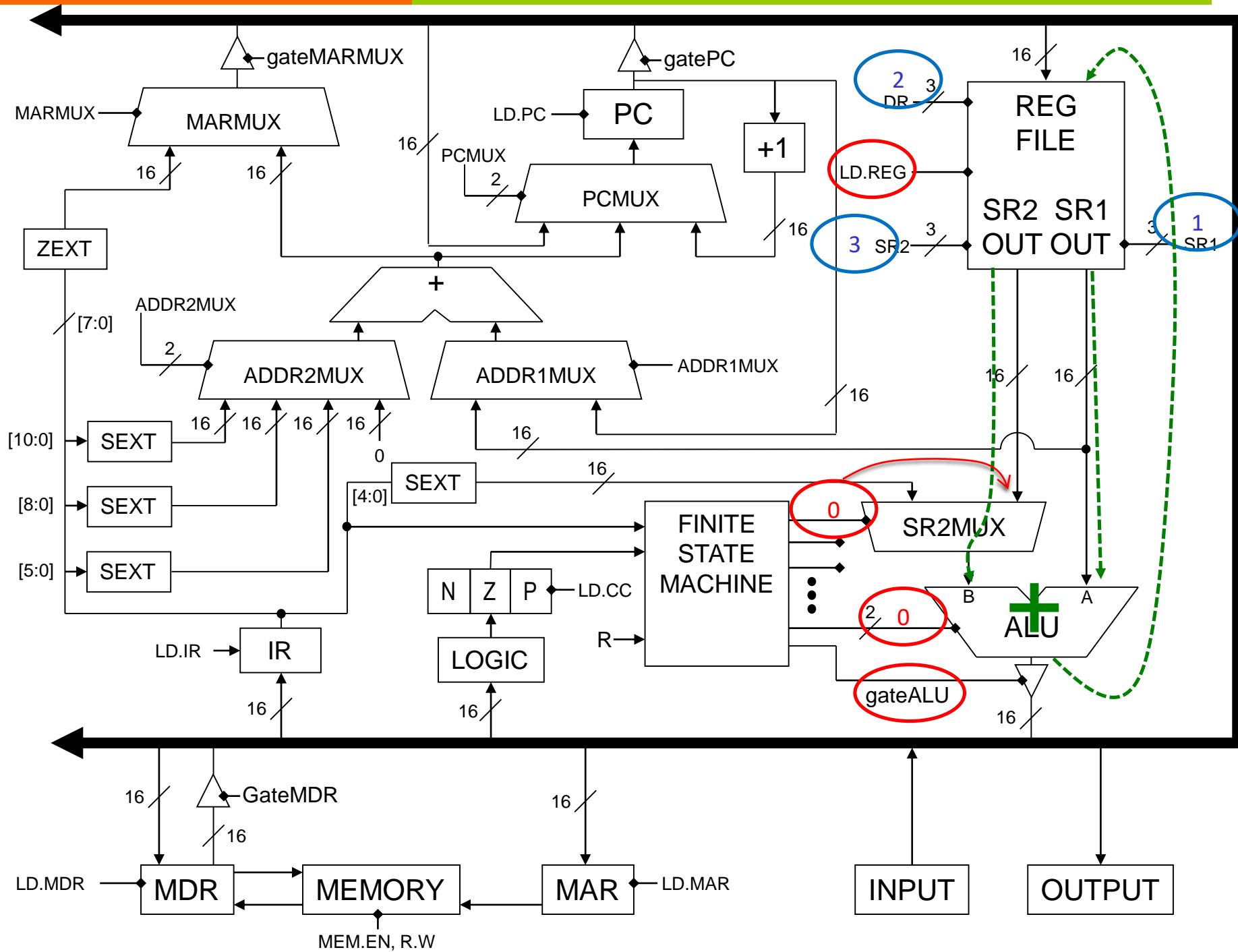
ADD R2, R1, R3 (Execute)

- Copy R1 to ALU Input A
- Copy R3 to ALU input B
- Set ALU to ADD



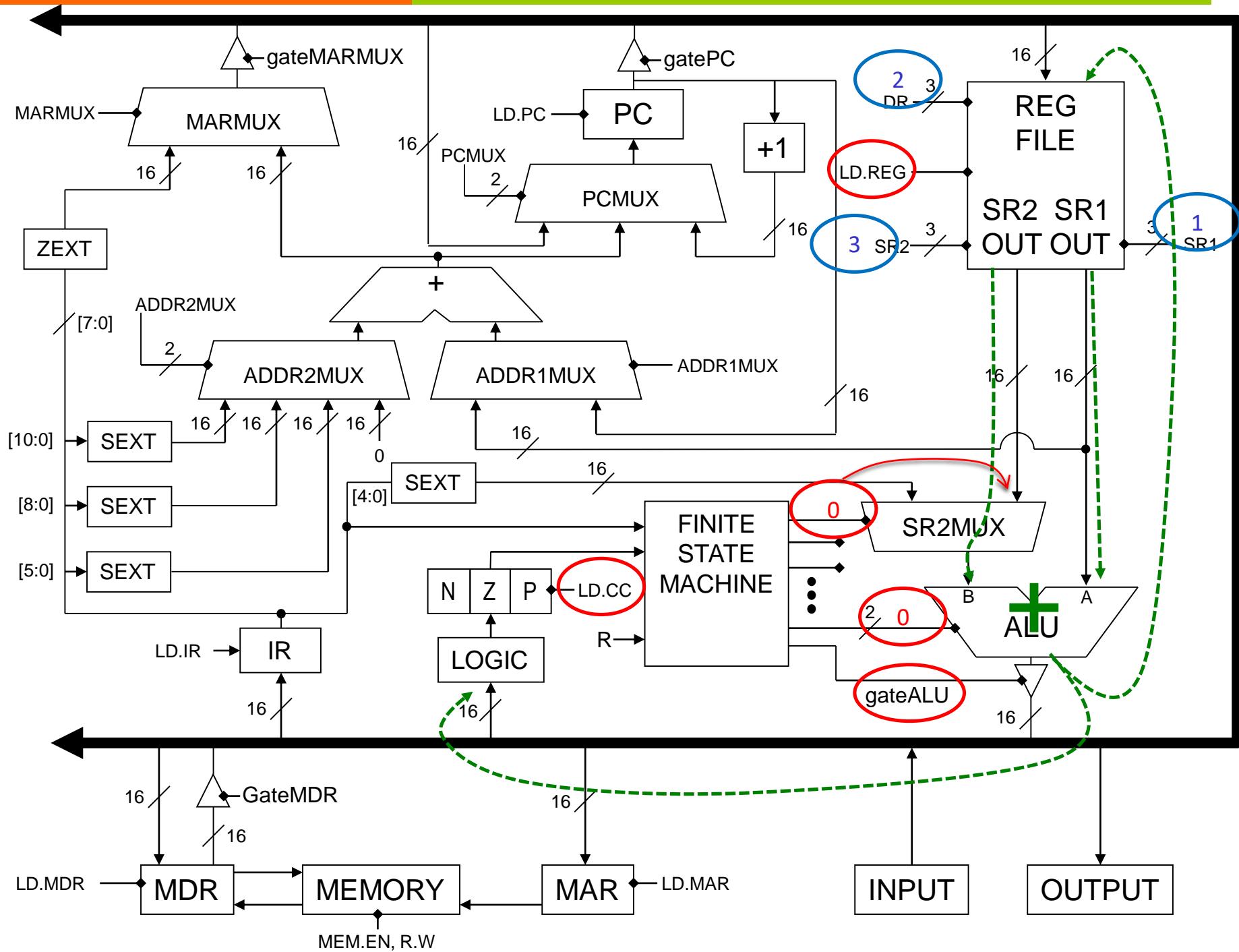
ADD R2, R1, R3 (Execute)

- Copy R1 to ALU Input A
- Copy R3 to ALU input B
- Set ALU to ADD
- **Copy ALU output to R2**



ADD R2, R1, R3 (Execute)

- Copy R1 to ALU Input A
- Copy R3 to ALU input B
- Set ALU to ADD
- Copy ALU output to R2
- And copy ALU output to CC



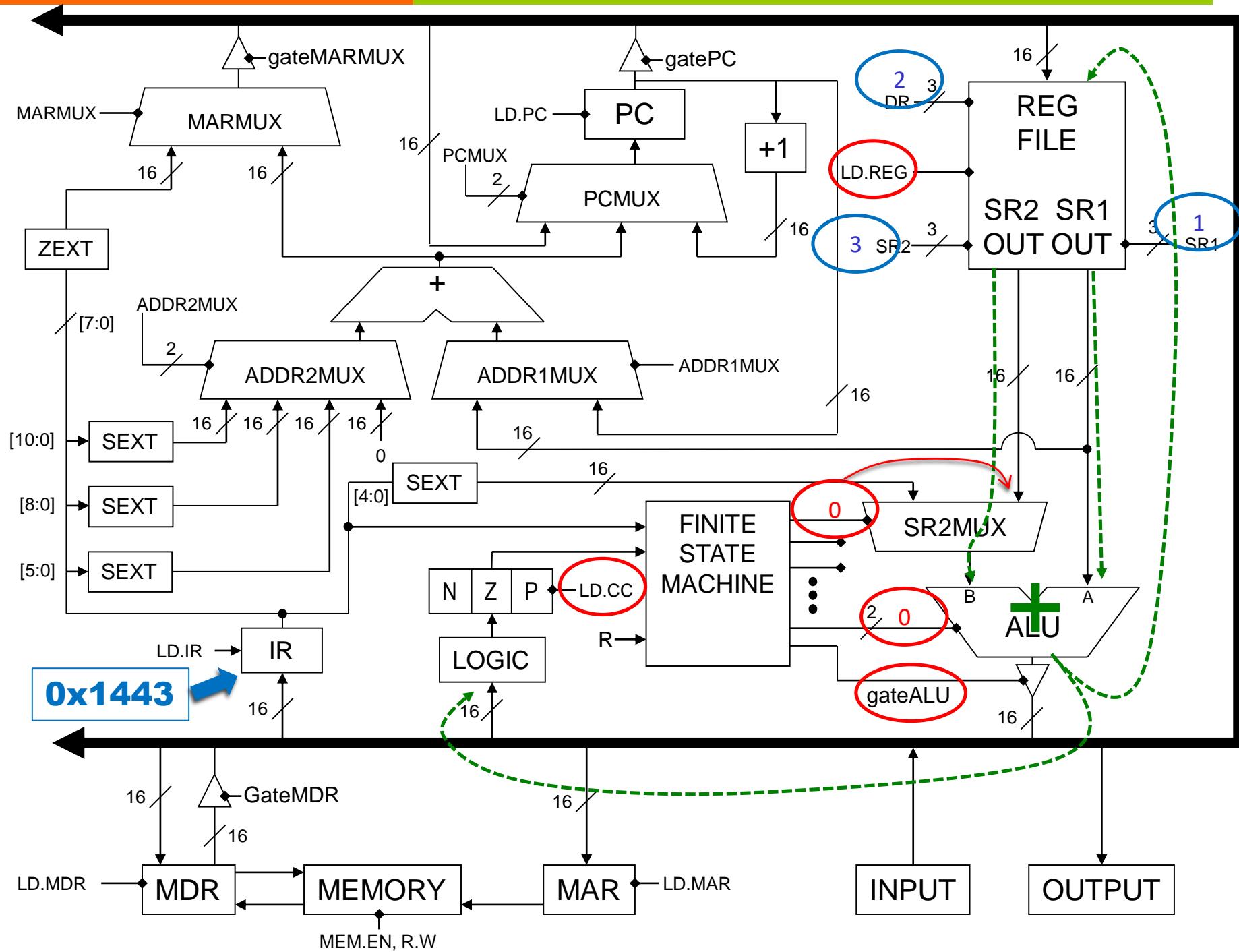
ADD R2, R1, R3 (Execute)

Control Signals
(from FSM):

- LD.REG=1
- gateALU=1
- SR2MUX=1
- ALUK=00
- LD.CC =1

From Instruction (IR):

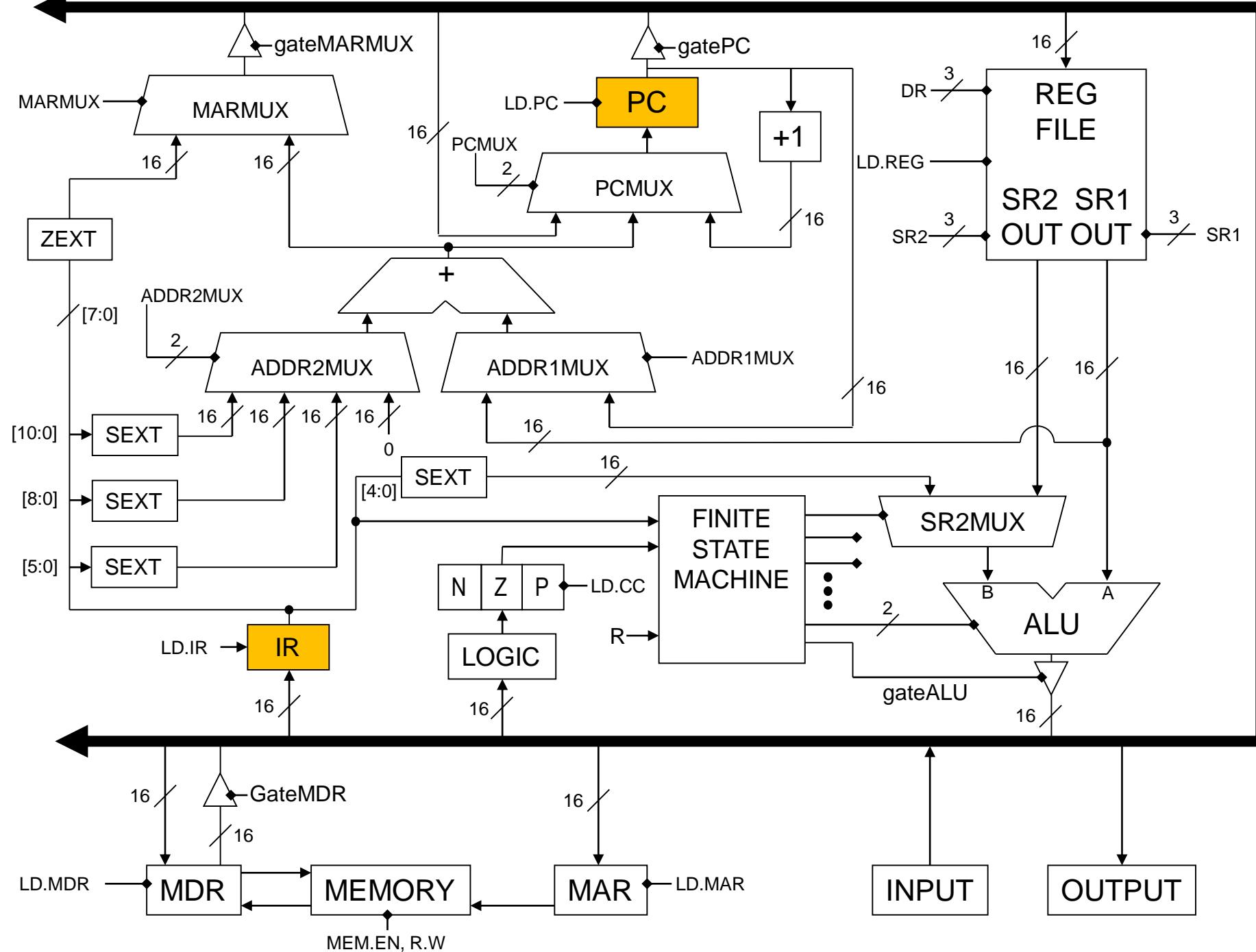
- SR1=001
- SR2=011
- DR=010



Exactly How Did the FSM Get That Add Instruction?

- Before the FSM could execute the ADD instruction in the previous slides, it had to fetch the instruction from memory
- The FSM has a procedure that it executes over-and-over to process instructions
 - FETCH
 - DECODE
 - EXECUTE
- The FSM fetches and decodes each instruction before it executes it
- We'll talk about the "fetch-execute" loop in detail in a bit

Back to the full LC-3 Datapath



What is a machine code program?

- A series of machine code instructions.
- Each instruction is a 16-bit data value.
- The instructions are stored in memory.
 - Usually in sequence
 - Consecutive memory locations
- Each memory location has an address
 - The address is a 16-bit unsigned integer.
- **Do you know what the PC and IR registers do, from the Patt book?**

Program Counter (PC)

The Program Counter (PC) register holds

- A. The count of instructions executed
-  B. The address of the next instruction to be executed
- C. The instruction currently being executed
- D. Number of programs completed since the last boot

Instruction Register (IR)

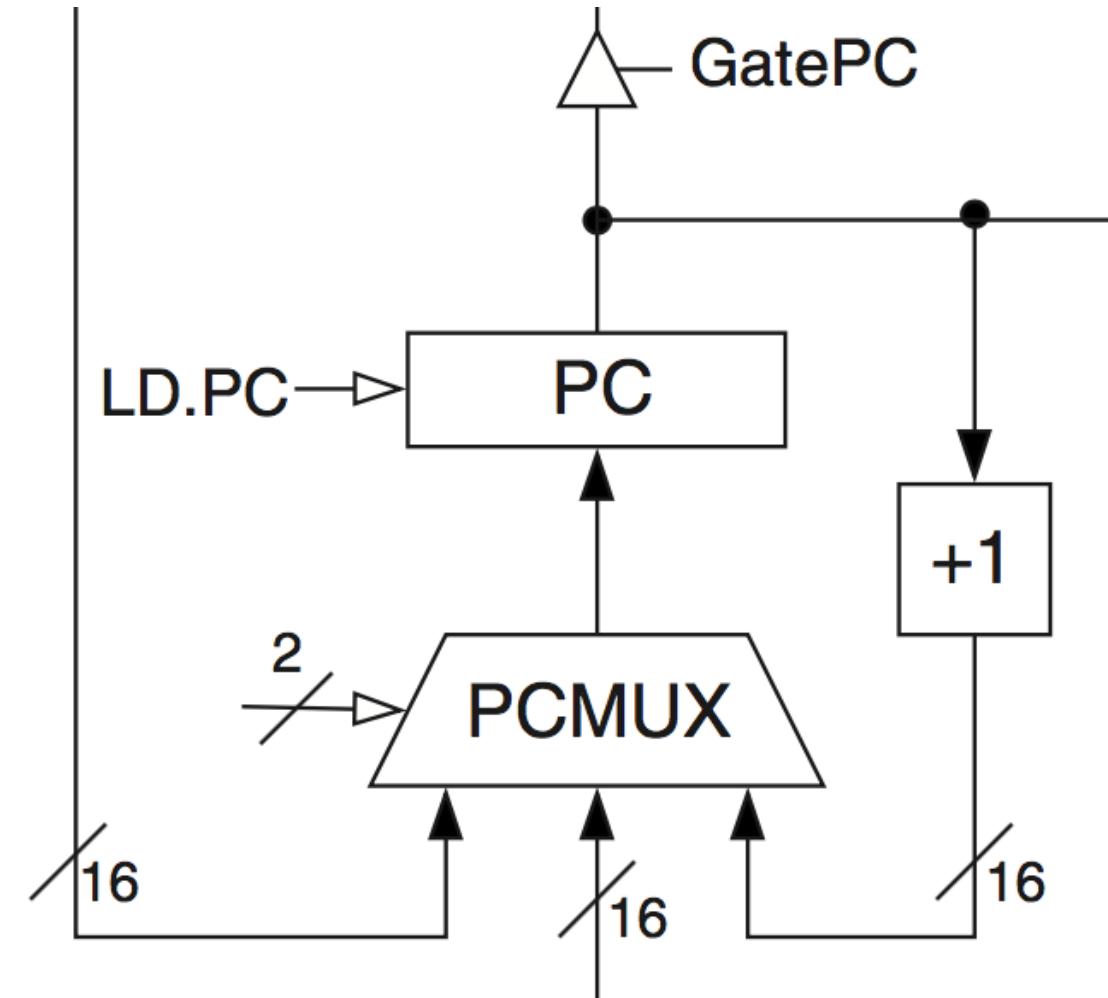
The Instruction Register (IR) holds

- A. All legal op codes
- B. The address of the next instruction to be executed
- C. All data needed to execute the current instruction
- D. The instruction currently being executed



Program Counter circuit

- PCMUX
 - Adder to increment PC (+1)
 - Data from bus to PC
 - Data from effective address* calculator to PC
- LD.PC
 - Write enable for PC reg
- GatePC
 - Put PC value on bus



* We'll explain effective address, and why you need it, later. But you know what a MUX is.

Control Signals for PC circuit

- GatePC (tri-state buffer)
- LD.PC (write enable for PC register)
- PC MUX (2 bit mux selector)

- 4 bits of control signals total

FETCH and DECODE

- The ADD machine code instruction – that we saw earlier – lives in memory (*in our machine code program*).
- Before we do the add (or any other instruction), we must get the machine code instruction from memory.
 - The PC register tells us the memory address where the instruction is located.
 - The IR holds the value read from memory (the machine code 16-bit instruction itself)

FETCH and DECODE

↗ FETCH

- ↗ Takes 3 clock cycles to read data (the instruction) from memory

DECODE

- ↗ Takes 1 clock cycle

- ↗ This is where the FSM generates the control signals for the specific instruction (such as ADD).

- ↗ Total: 4 clock cycles (states) to fetch and decode an instruction.

- ↗ This happens at the beginning of EVERY machine code instruction.

Machine Code Instruction Cycle

Progression through States

- ↗ Fetch
- ↗ Decode
- ↗ Evaluate Address [*optional*]
- ↗ Fetch Operands [*optional*]
- ↗ Execute [*optional*]
- ↗ Store Result [*optional*]



ADD instruction only uses Fetch, Decode, Execute

- ↗ Fetch
- ↗ Decode
- ↗ Evaluate Address [*optional*]
- ↗ Fetch Operands [*optional*]
- ↗ Execute [*optional*]
- ↗ Store Result [*optional*]



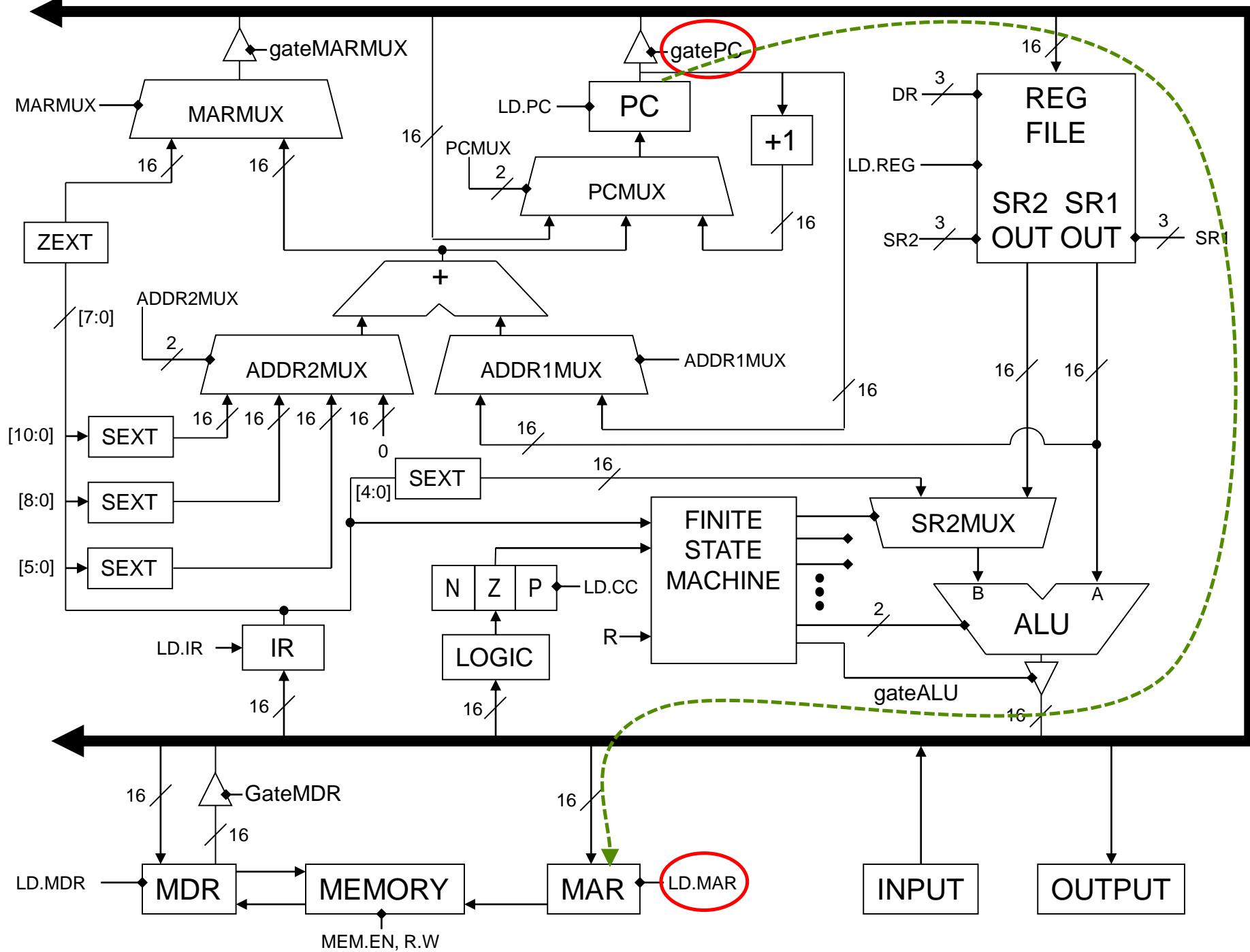
Instruction Processing: FETCH

- Load next instruction (at address stored in PC) from memory into Instruction Register (IR).
 - Copy contents of PC into MAR.
 - Send “read” signal to memory.
 - Copy contents of MDR into IR.
- Increment PC, so that it points to the next instruction in sequence.
 - PC becomes PC+1.



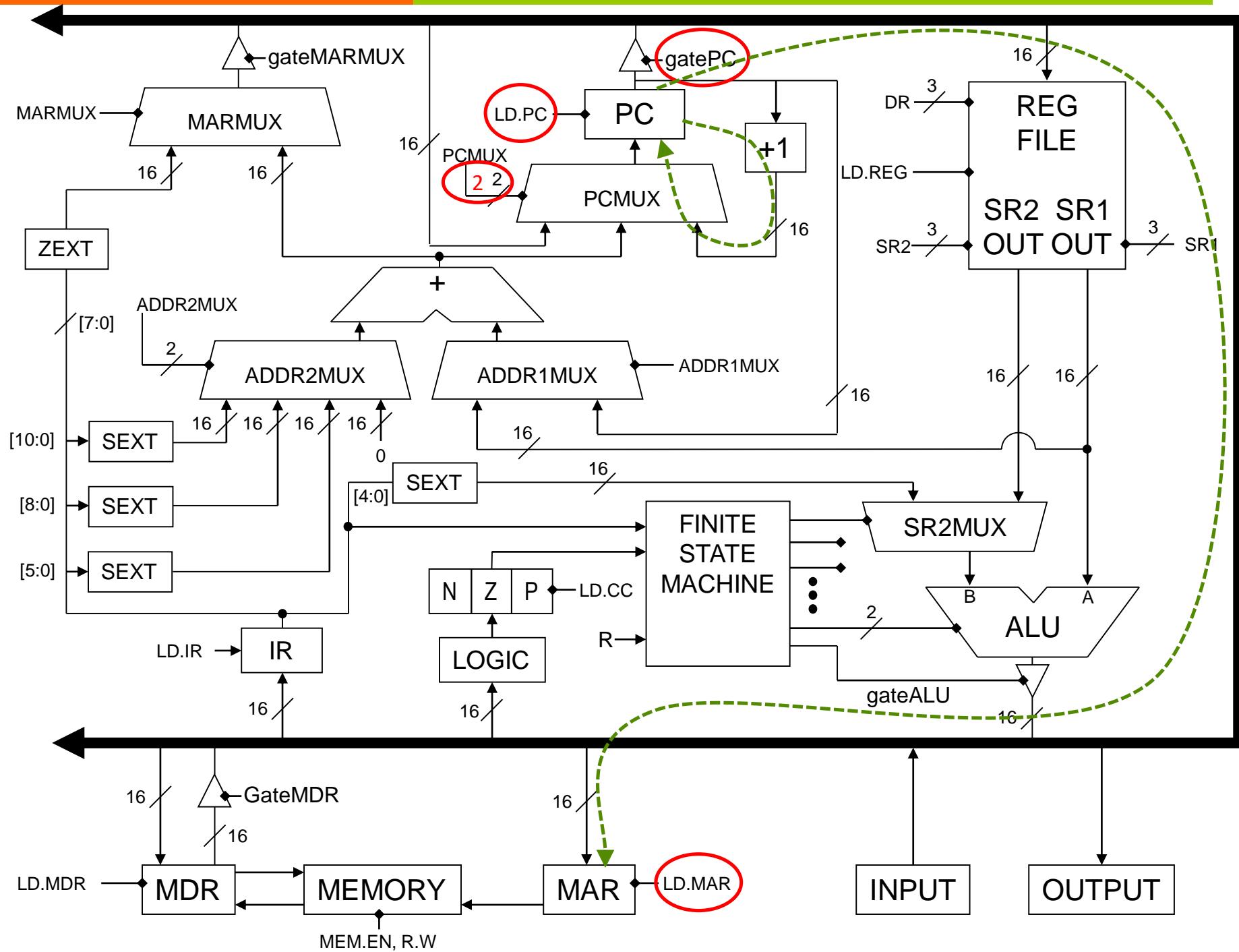
FETCH (Phase 1)

- Copy PC to MAR



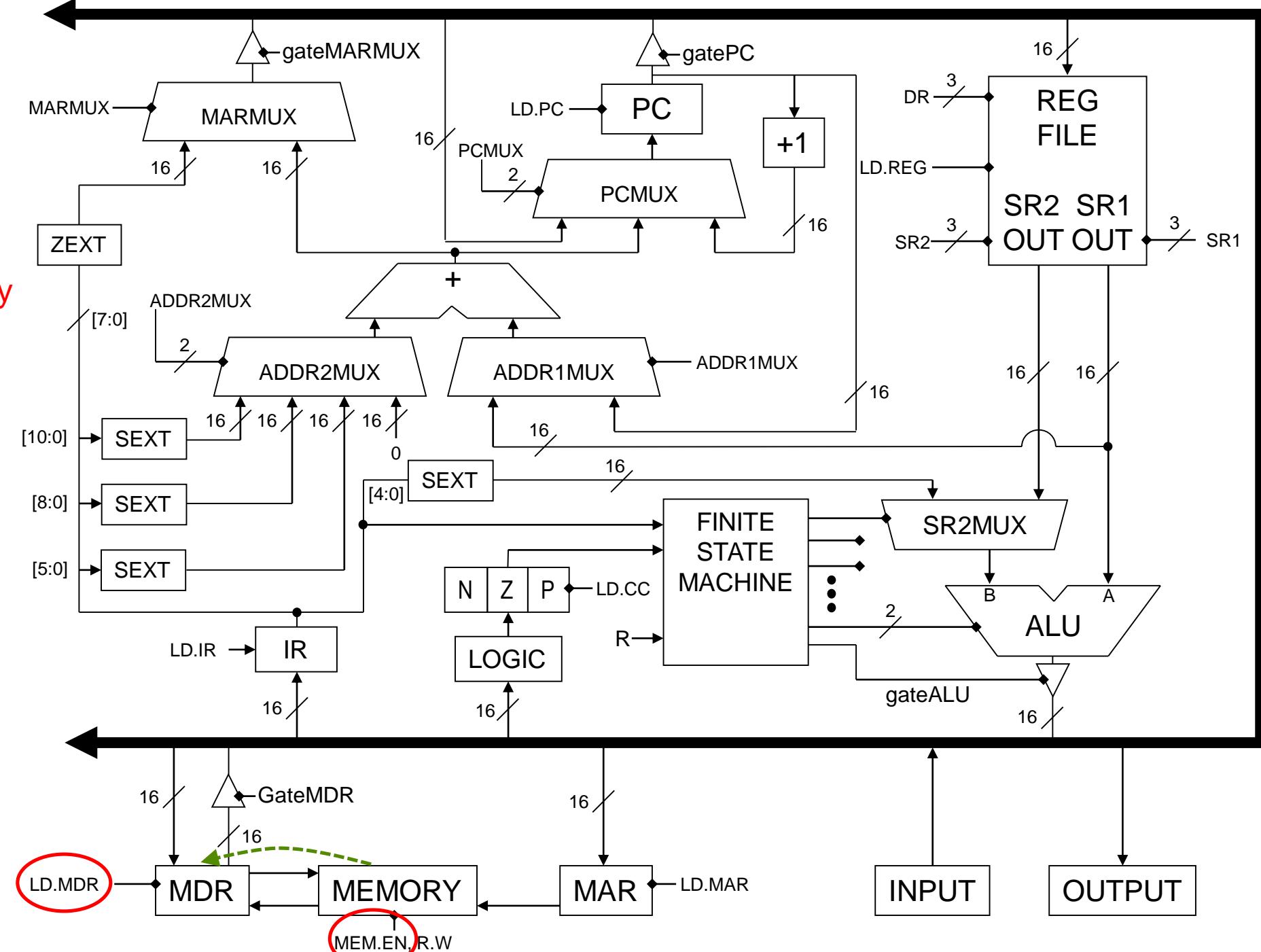
FETCH (Phase 1)

- Copy PC to MAR
- Increment PC



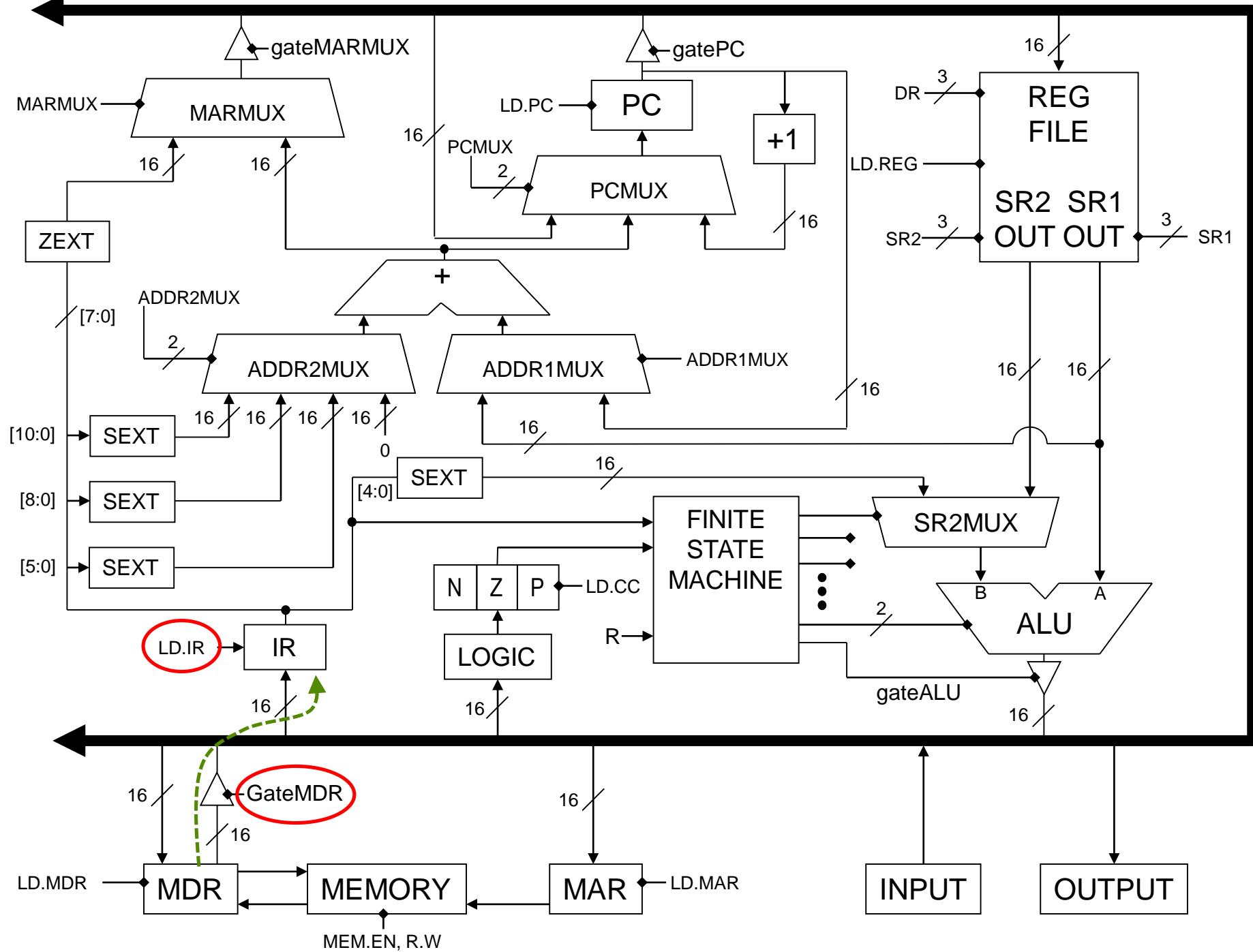
FETCH (Phase 2)

- Read from memory into MDR



FETCH (Phase 3)

- Copy MDR into IR



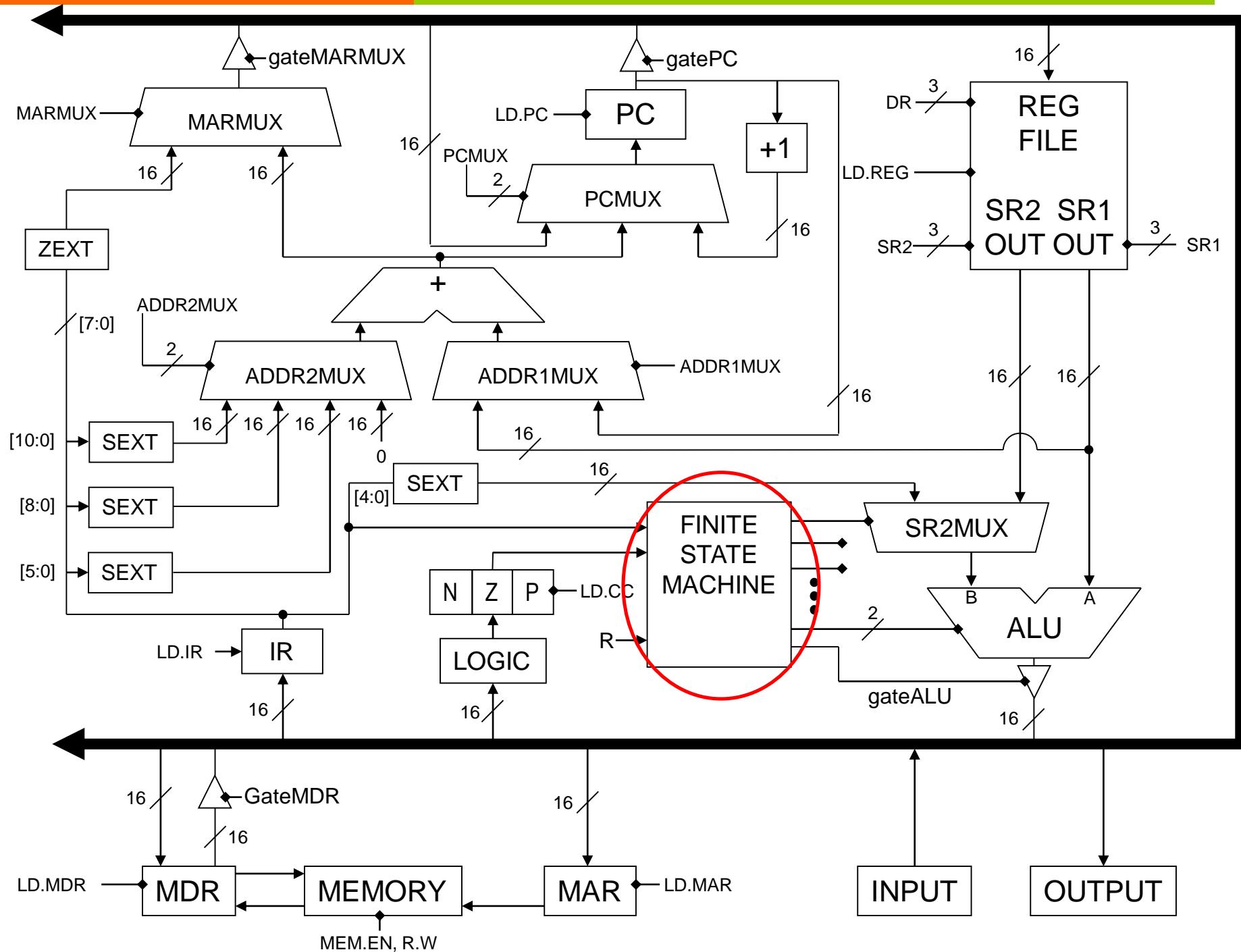
Instruction Processing: DECODE

- First identify the opcode.
 - On the LC-3, this is 4 bits [15:12].
 - The FSM chooses a next state corresponding to the desired opcode.
- Depending on opcode, identify other operands from the remaining bits.
 - Example:
 - for LDR, last six bits is offset
 - for ADD, last three bits is source operand #2
- DECODE is the Finite State Machine!
 - It activates the control signals in the datapath.
 - We'll discuss FETCH in detail later.



DECODE

- FSM



Instruction Processing: EXECUTE

- Perform the operation, using the source operands.
- Examples:
 - send operands to ALU and assert ADD control signal
 - do nothing (e.g., for loads and stores to memory – more on this later)



EXECUTE

- ↗ It depends
 - ↗ Every instruction has a different execute phase
 - ↗ With different control signals
- ↗ But every instruction starts with:
 - ↗ FETCH
 - ↗ FETCH
 - ↗ FETCH
 - ↗ DECODE

Instruction Processing Review

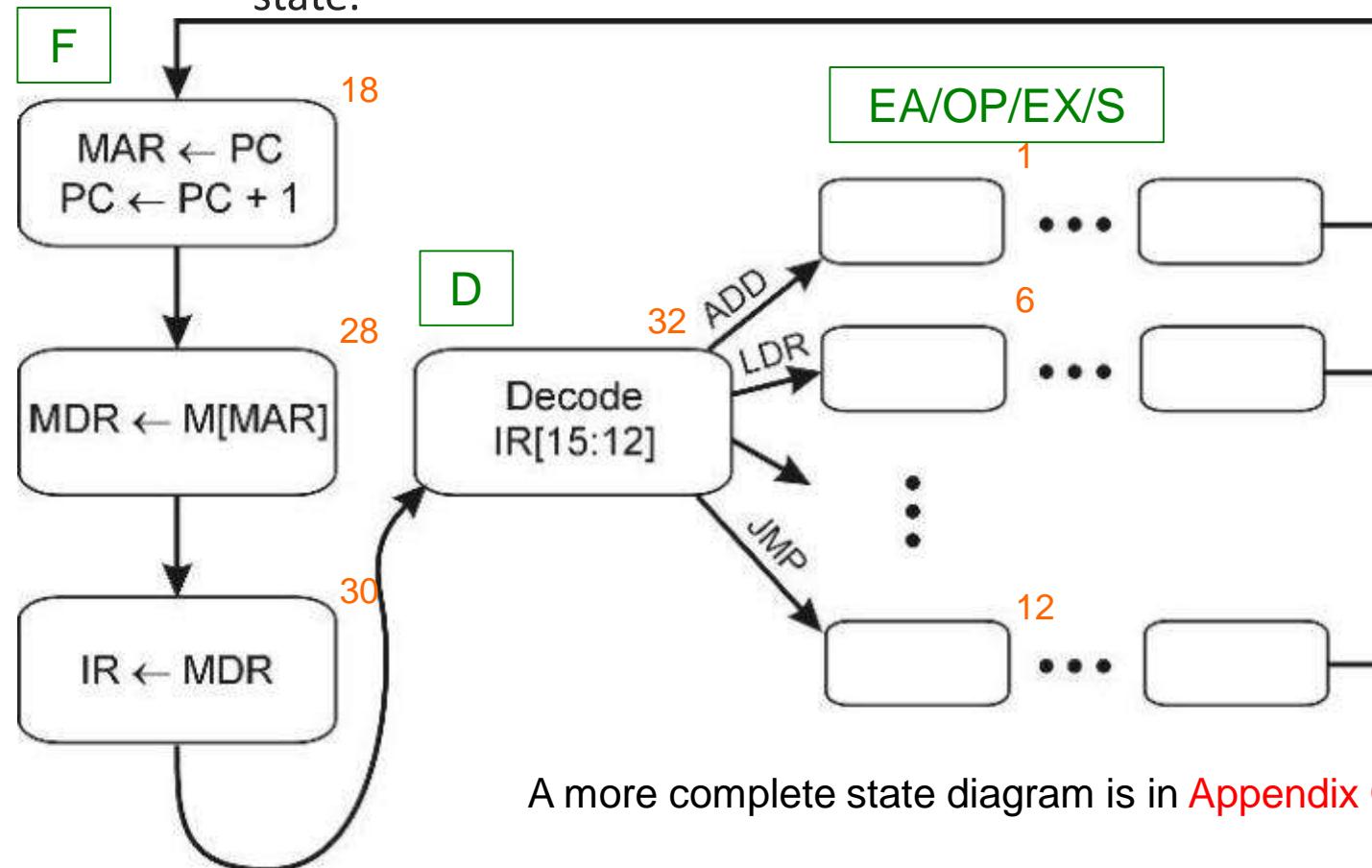
- ↗ Instruction bits look just like data bits in memory – it's all a matter of our interpretation
- ↗ Three basic kinds of instructions:
 - ↗ computational instructions (ADD, AND, NOT)
 - ↗ data movement instructions (LD, ST, ...)
 - ↗ control instructions (JMP, BRnz, ...)
- ↗ Six basic phases of instruction processing:
 $F \rightarrow D \rightarrow EA \rightarrow OP \rightarrow EX \rightarrow S$
 - ↗ not all phases are needed by every instruction
 - ↗ phases may take variable number of machine cycles (states)

The LC-3 is a Finite State Machine

- ↗ LC-3 Finite State Machine
 - ↗ Has 64 possible states
 - ↗ Orchestrates 42 control signals
 - ↗ Multiplexor selectors
 - ↗ PCMUX, MARMUX, ADDR2MUX, ...
 - ↗ Tri-state buffer enables
 - ↗ gatePC, gateMARMUX, gateALU, ...
 - ↗ Register write-enables
 - ↗ LD.PC, LD.REG, LD.MAR, LD.CC, ...
 - ↗ Other control signals
 - ↗ ALUK, MEM.EN, R.W, ...
 - ↗ The wires aren't all shown explicitly on the datapath diagram – *to avoid clutter*, but each signal has a name

What Makes the Fetch-Execute Cycle Happen?

- The FSM, which sequences through the states.
- It turns on the appropriate control signals as it enters each state.



A more complete state diagram is in [Appendix C Fig C.3](#)

The LC-3 is a Finite State Machine

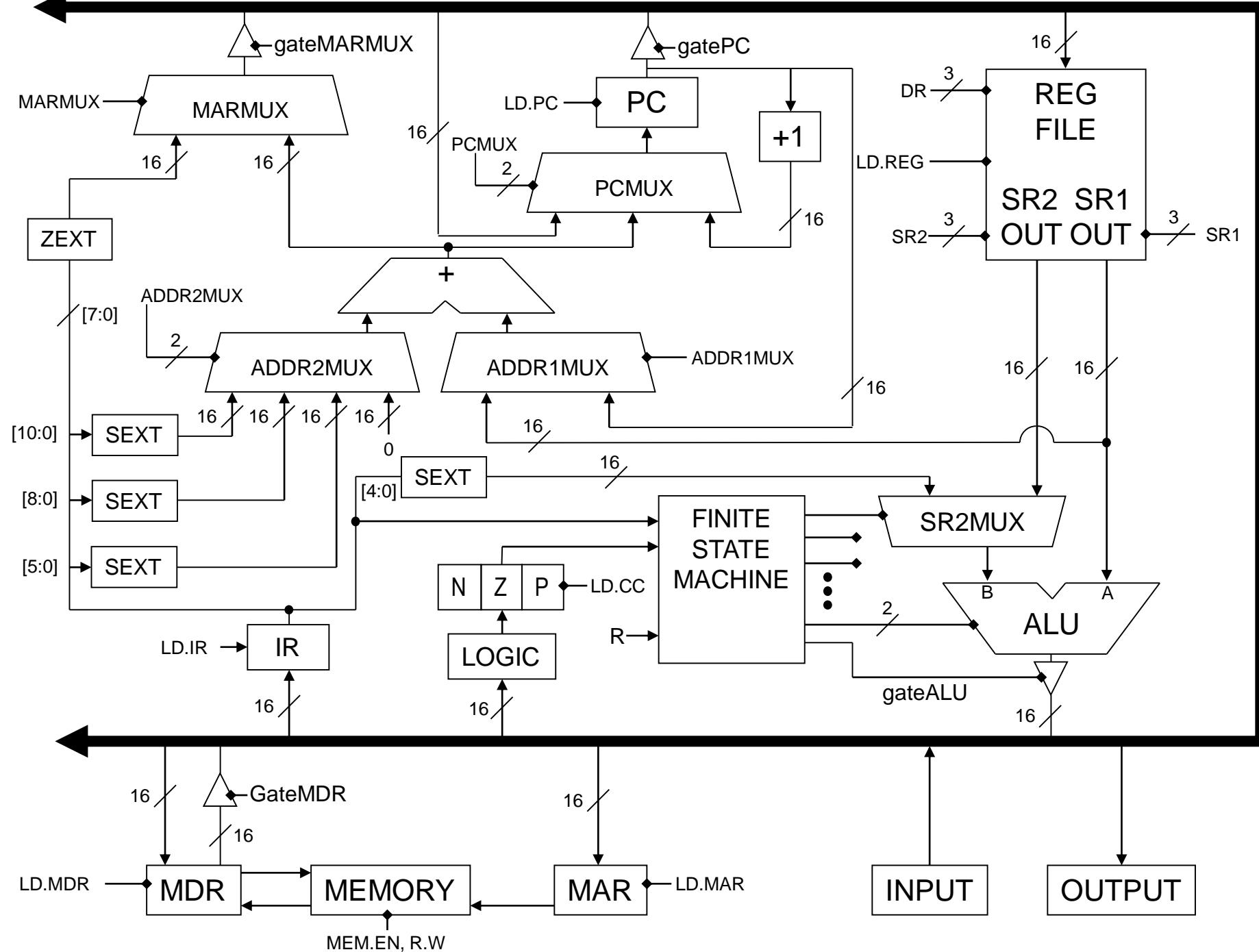
- ↗ How do we implement the LC-3 state machine?
 - ↗ Just like the garage door opener, we build logic to produce output control signals.
 - ↗ The garage door had 2 output signals: UP and DOWN.
- ↗ The LC-3 has 42 control signals.
 - ↗ These signals control the datapath (the MUX selectors, tri-state buffers, write enables (e.g LD.REG), etc.
 - ↗ The output of the FSM is the 42 control signals.
- ↗ The LC-3 is a glorified state machine.
 - ↗ But works just like the garage door opener.
 - ↗ It has 64 states, and 42 control signal outputs.

When the Fetch-Execute cycle reaches the EXECUTE phase for an instruction

- A. The PC contains the address of the next instruction to be *Why not?*
- B. The PC contains the address of the instruction being executed
- C. The PC contains the address of the word after the instruction
being executed
- D. The PC contains the instruction being executed



Today's number is 48,360



- Basic CPU Components
 - Memory, Processing Unit, Input & Output, Control Unit
- Architectures
 - Harvard
 - Von Neumann
- LC-3: A von Neumann Architecture
- Tri-State Buffers
- Instruction Cycle
 - Instructions
 - Fetch, Decode, Evaluate Address, Fetch Operands, Execute, Store Result

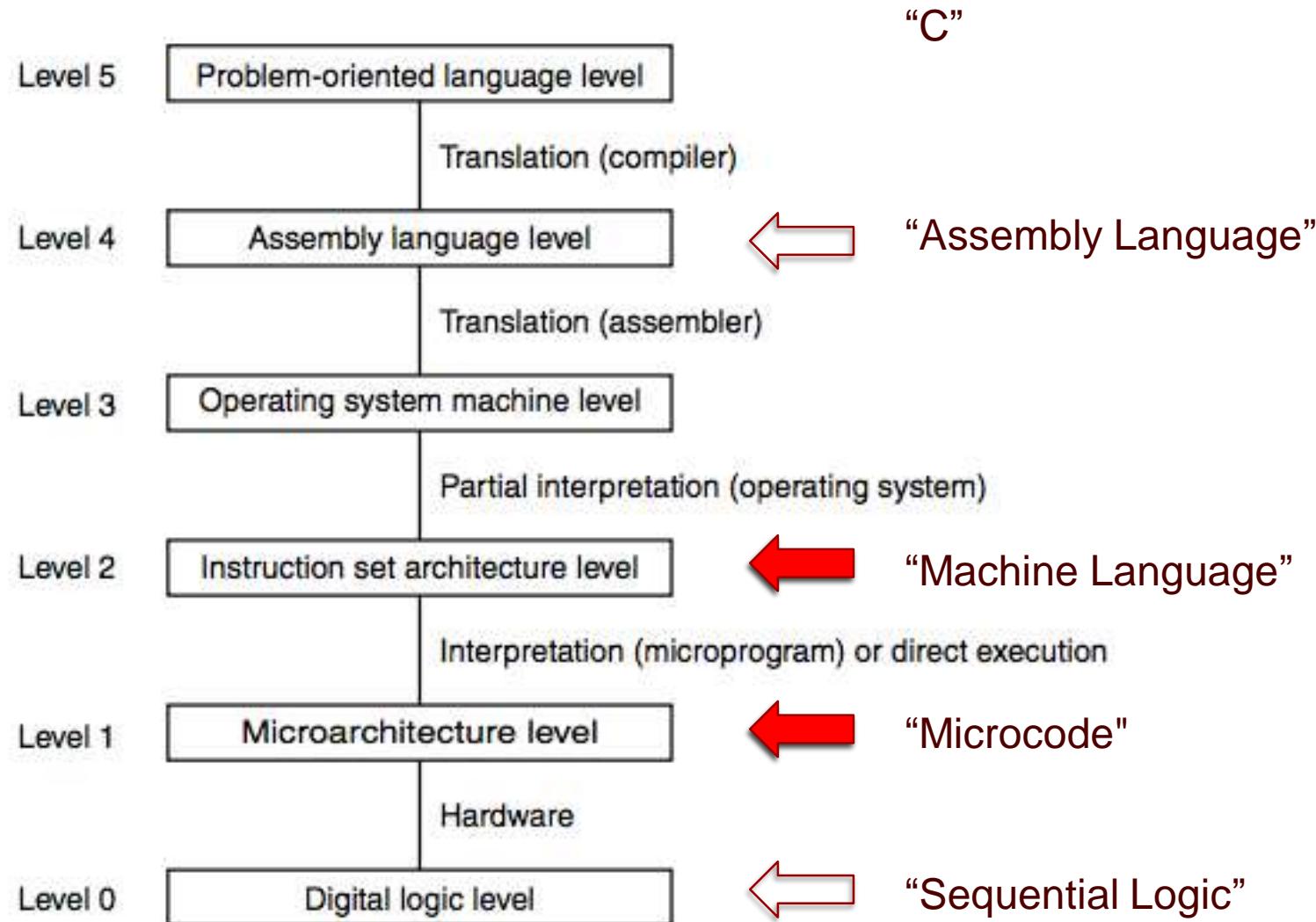
The LC-3 Part I

Chapter 5



- The ISA: Overview
 - Memory, Registers, Instruction Set, Opcodes, Datatypes, Addressing Modes, Condition Codes
- Instruction Set
 - Datapath, Control signals to implement each instruction
 - ALU, Data movement (load/store), and Control Instructions
 - Addressing Modes
- The Microsequencer and Microcode

Road Map -- Where are we?



Instruction Set Architecture

- ↗ The ISA specifies all the information about the computer that the software needs to be aware of.
 - ↗ The ISA is a programmer's view of the CPU (e.g. LC-3) from the outside (*how do I write machine code/assembly?*)
 - ↗ The internal implementation of the LC-3 is the datapath and microcode.
- ↗ Who uses an ISA?
- ↗ What is specified?

Who uses an ISA?

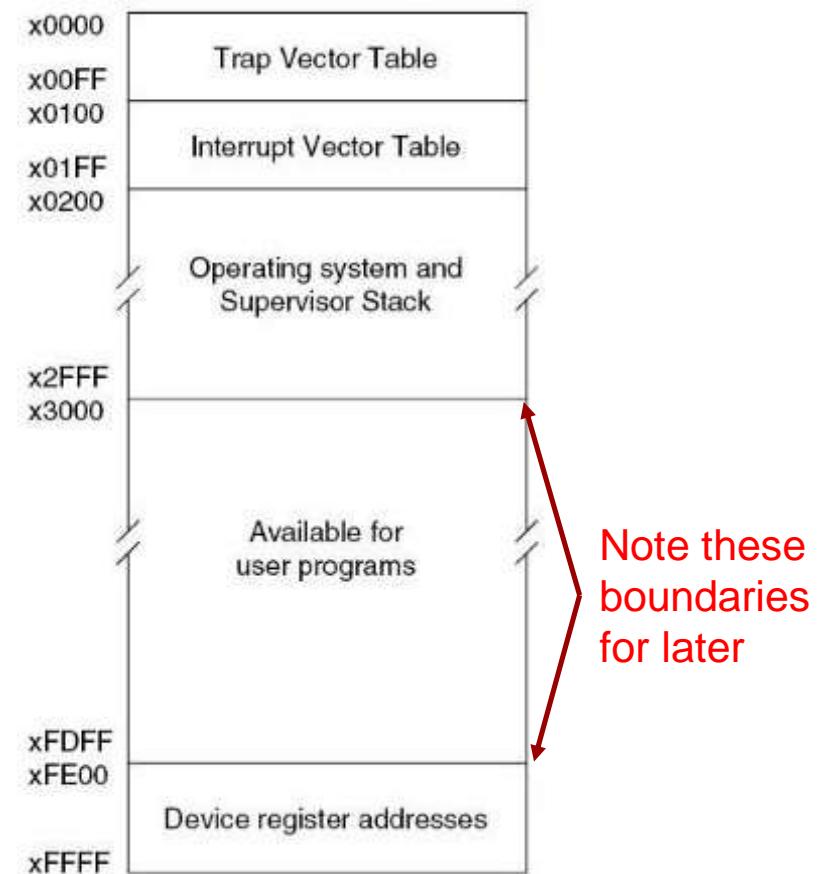
- ↗ Machine language programmers
- ↗ Assembly language programmers
- ↗ Compiler writers
- ↗ High-level language programmers interested in performance, debugging, etc.

ISA: What Is Specified?

- Memory organization
- Registers
- Instruction set
 - Opcodes
 - Data types
 - Addressing modes

LC-3 Memory Organization

- Address space
 - 16 bit addresses
 - 0-65535
 - x0-xFFFF
- Addressability
 - 16 bits
- Word Addressable/Byte Addressable?
 - Strictly word addressable with 16 bit words.



- General Purpose Registers

- Is the PC considered to be a GPR?

- Question:

- 1) Yes
 - 2) No

- Conventions

LC-3 Registers

- General Purpose Registers
 - 8 General Purpose Registers
 - R0 - R7
- Is the PC considered to be a GPR?
 - No
- Conventions
 - Certain Registers will have designated purposes
 - (That comes later; for now you can use all 8 of them)

Question

What memory addresses are allocated to store user programs and data?

- A. 0x2000 – 0x2FFF
- B. 0x2000 – 0xFE00
- C. 0x0000 – 0xFFFF
- D. 0x3000 – 0xFDFF



LC-3 Instruction Set

- Instruction
 - Op Code
 - Operands
- Instruction Set
 - Op Codes
 - Operate (ALU)
 - Data movement (load/store memory)
 - Control (conditionals, loops, etc.)
 - Data Types
 - Addressing Modes

How Many Instructions? (ISA design philosophies)

↗ Lots

- ↗ **CISC** (Complex Instruction Set Computer)
- ↗ Intel X86, DEC VAX, IBM Z-series
- ↗ Do as much as you can in a single instruction.
- ↗ Relatively easy to write efficient machine code by hand.

↗ Few

- ↗ **RISC** (Reduced Instruction Set Computer)
- ↗ ARM, PowerPC, MIPS, **LC-3**
- ↗ Expose as much as you can to the compiler so that it can optimize.
- ↗ Hard to write efficient machine code by hand.

LC-3 Machine Code and Assembly Language

- ↗ For now, we are focused on the **LC-3 datapath**
 - ↗ The circuit implementation, and the state machine
 - ↗ The control signals that make each instruction happen
- ↗ You will need to understand what each LC-3 machine code instruction does – *atomically, in isolation*
 - ↗ What registers or memory locations does it change?
 - ↗ Which ALU operation is happening?
 - ↗ How does the data flow through each MUX and the bus to make the operation execute with the desired behavior?
- ↗ You do NOT need to understand why or how you would use these instructions in a program yet.
 - ↗ We'll cover that later, with assembly programming
 - ↗ For now, just understand each instruction's behavior – so you can know how to implement it in the datapath and microcode, using control signals

Meet the LC-3 Instruction Set

- 16-bit instructions
- 4 bits for opcode (bits 15-12)
- How many instructions?
- 15 – code 1101 is reserved for future use.

➤ What is it?

- A symbolic (text, human readable) representation of the binary machine language
- Instructions are one-to-one between AL and ML

Example assembly language:

ADD R1, R2, R3

Categories of LC-3 Instructions

Operate (ALU)	Data Movement (Memory)	Control
<ul style="list-style-type: none">• ADD• AND• NOT	<p><u>Load</u></p> <ul style="list-style-type: none">• LD• LDR• LDI• LEA <p><u>Store</u></p> <ul style="list-style-type: none">• ST• STR• STI	<ul style="list-style-type: none">• BR• JMP• JSR• JSRR• RET• RTI• TRAP

Operate (ALU) Instructions

Operate (ALU)

- ADD
- AND
- NOT

Data Movement (Memory)

Load

- LD
- LDR
- LDI
- LEA

Store

- ST
- STR
- STI

Control

- BR
- JMP
- JSR
- JSRR
- RET
- RTI
- TRAP

LC-3 Instructions, first half

ADD+	0001	DR	SR1	0	00	SR2
ADD+	0001	DR	SR1	1	imm5	
AND+	0101	DR	SR1	0	00	SR2
AND+	0101	DR	SR1	1	imm5	
BR	0000	n	z	p	PCoffset9	
JMP	1100	000	BaseR	000000		
JSR	0100	1		PCoffset11		
JSRR	0100	0	00	BaseR	000000	
LD+	0010	DR		PCoffset9		
LDI+	1010	DR		PCoffset9		

+ Indicates instructions that modify condition codes

LC-3 Instructions, second half

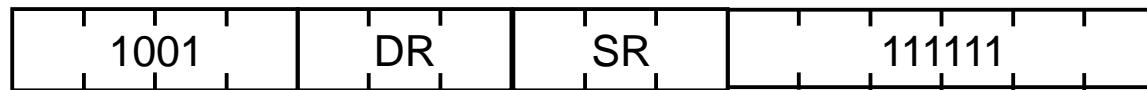
LDR+	0110	DR	BaseR	offset6
LEA	1110	DR		PCoffset9
NOT+	1001	DR	SR	111111
RET	1100	000	111	000000
RTI	1000			000000000000
ST	0011	SR		PCoffset9
STI	1011	SR		PCoffset9
STR	0111	SR	BaseR	offset6
TRAP	1111	0000		trapvect8
reserved	1101			

+ Indicates instructions that modify condition codes

Let's Look at the NOT Instruction

- ↗ NOT

NOT



- ↗ The operand (SR) and the result (DR) are both stored in registers
 - ↗ in the register file (R0-R7)
- An example NOT instruction:

NOT R3,R3

1001

011

011

111111

Machine Instruction Cycle

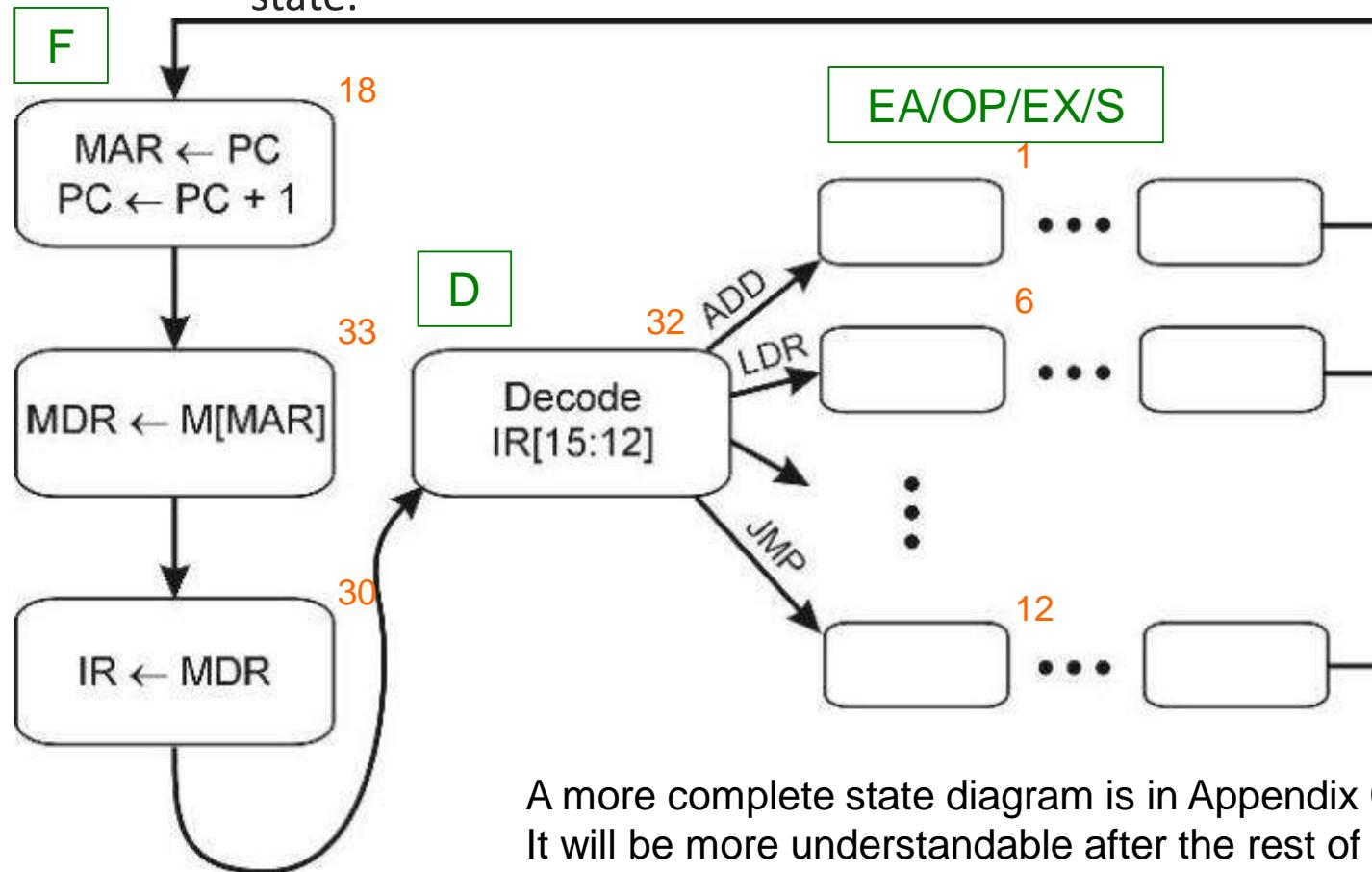
➤ Recall our states:

1. Fetch 1
2. Fetch 2
3. Fetch 3
4. Decode
5. Execute

➤ Fetch and Decode are the same for EVERY instruction.

What Makes the Fetch-Execute Cycle Happen?

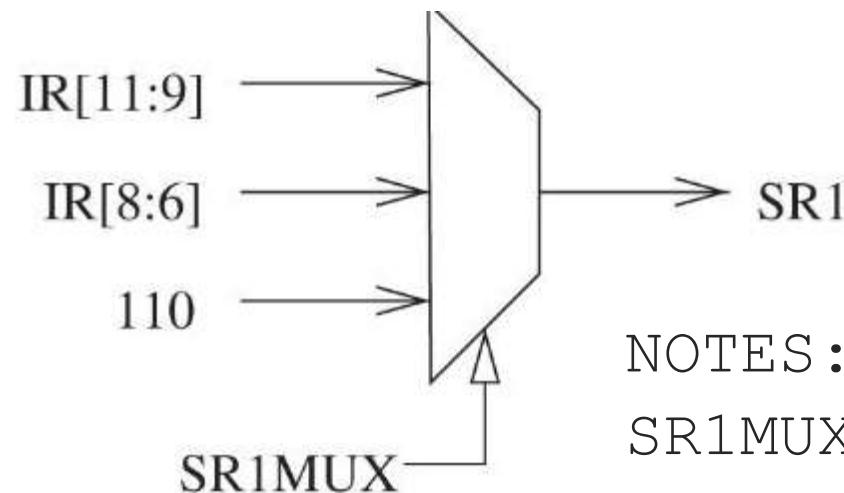
- The FSM, which sequences through the states.
- It turns on the appropriate control signals as it enters each state.



EXECUTE State for NOT

001001 (State 9) 0 0 J=18

LD.REG LD.CC GateALU SR1MUX=1 ALUK=2
all other control signals are 0



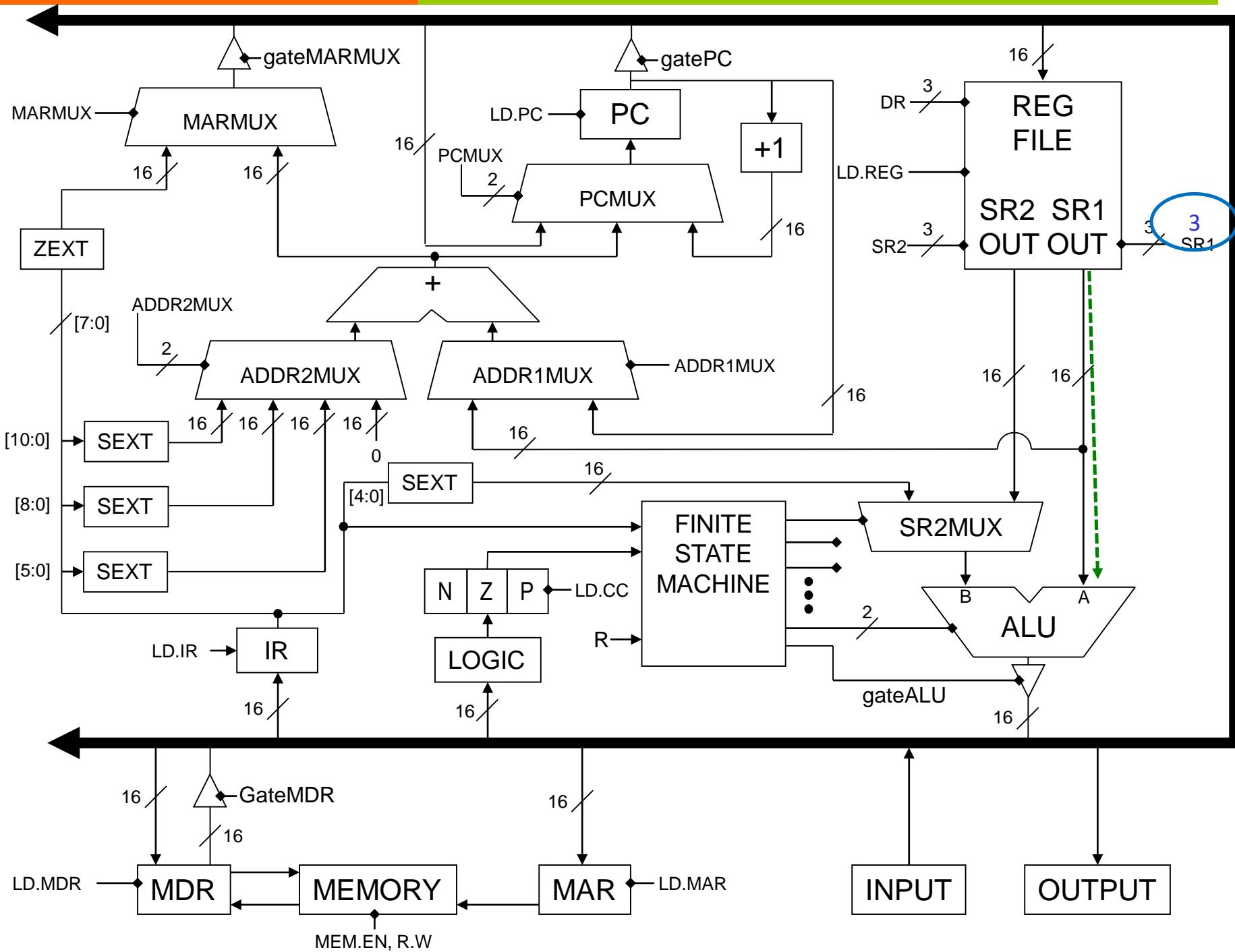
NOTES:

SR1MUX=1 means use IR[8:6] as SR1
(see Fig C.6)

ALUK=2 means "NOT A"

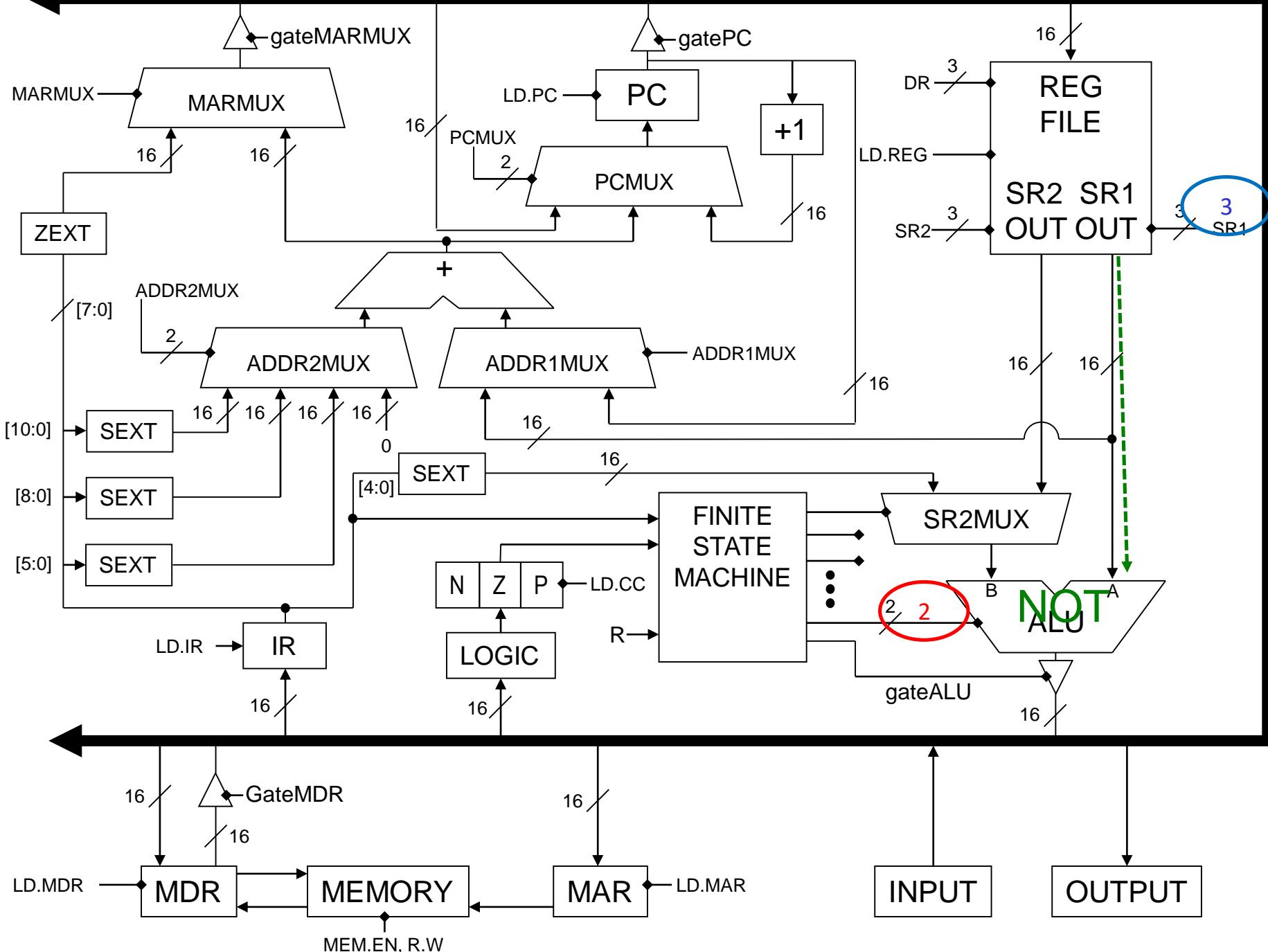
NOT R3, R3 (Execute)

- Copy R3 to ALU Input A



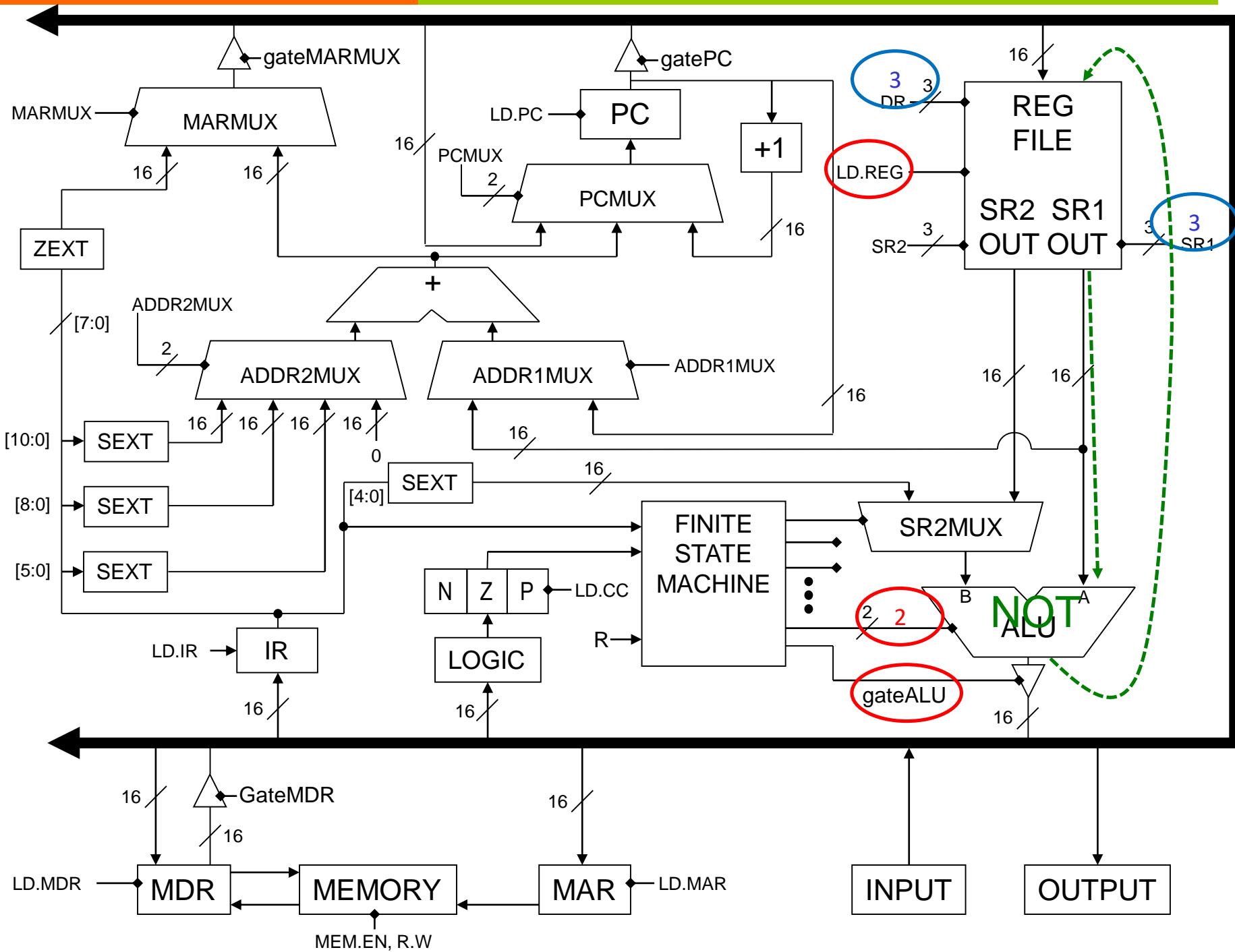
NOT R3, R3 (Execute)

- Copy R3 to ALU Input A
- Set ALU to NOT



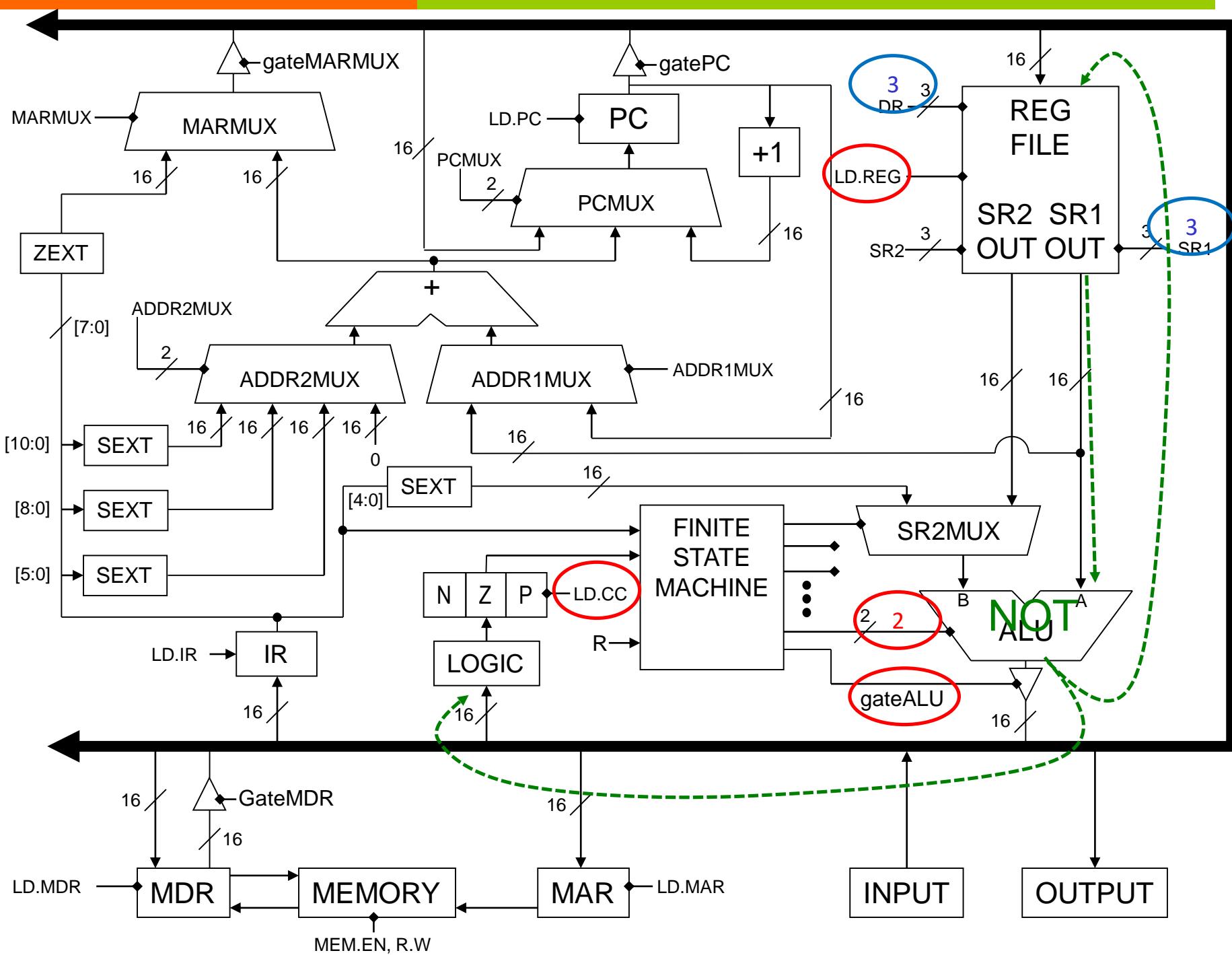
NOT R3, R3 (Execute)

- Copy R3 to ALU Input A
- Set ALU to NOT
- **Copy ALU output to R3**



NOT R3, R3 (Execute)

- Copy R3 to ALU Input A
- Set ALU to NOT
- Copy ALU output to R3
- And copy ALU output to CC



Question

How many cycles does it take for the EXECUTE phase of a NOT instruction

- A. 1 
- B. Either 1 or 2
- C. 2
- D. 3

Operate (ALU) Instructions

↗ ADD, AND

		Addressing Mode						
	ADD	0001	DR	SR1	0	00	SR2	Register
	ADD	0001	DR	SR1	1	imm5	Immediate or Literal	
	AND	0101	DR	SR1	0	00	SR2	Register
	AND	0101	DR	SR1	1	imm5	Immediate or Literal	

↗ Addressing Mode?

- ↗ Where do the operands come from?
- ↗ Depends on bit 5

Addressing Modes (for ALU operations)

- ↗ Register:
 - ↗ All operands come from register file
 - ↗ Examples:
 - ↗ ADD R1, R2, R3
 - ↗ AND R1, R2, R3
- ↗ Immediate or Literal
 - ↗ Some operands come from bits in the instruction itself (get them from the IR, Instruction Register)
 - ↗ Immediate values are 5-bit two's complement (for ADD and AND)
 - ↗ From bits [0:4] in the instruction itself
 - ↗ Sign-extend the immediate value to 16 bits two's complement
 - ↗ Examples:
 - ↗ ADD R1, R2, #3
 - ↗ AND R1, R2, #-3

EXECUTE States for ADD & AND

ADD

000001 (State 1) 0 0 J=18
LD.REG LD.CC GateALU SR1MUX=1

AND

000101 (State 5) 0 0 J=18
LD.REG LD.CC GateALU SR1MUX=1 ALUK=1



NOTES:

All other control signal are 0

SR1MUX=1 means use IR[8:6] as SR1 (see Patt)
SR2MUX select is IR[5]

ALUK=0 means "ADD A and B" (*not shown above, default is 0*)

ALUK=1 means "AND A and B"

Example

ADD R3, R3, R2

0001 011 011 000 010

Register
Addressing Mode

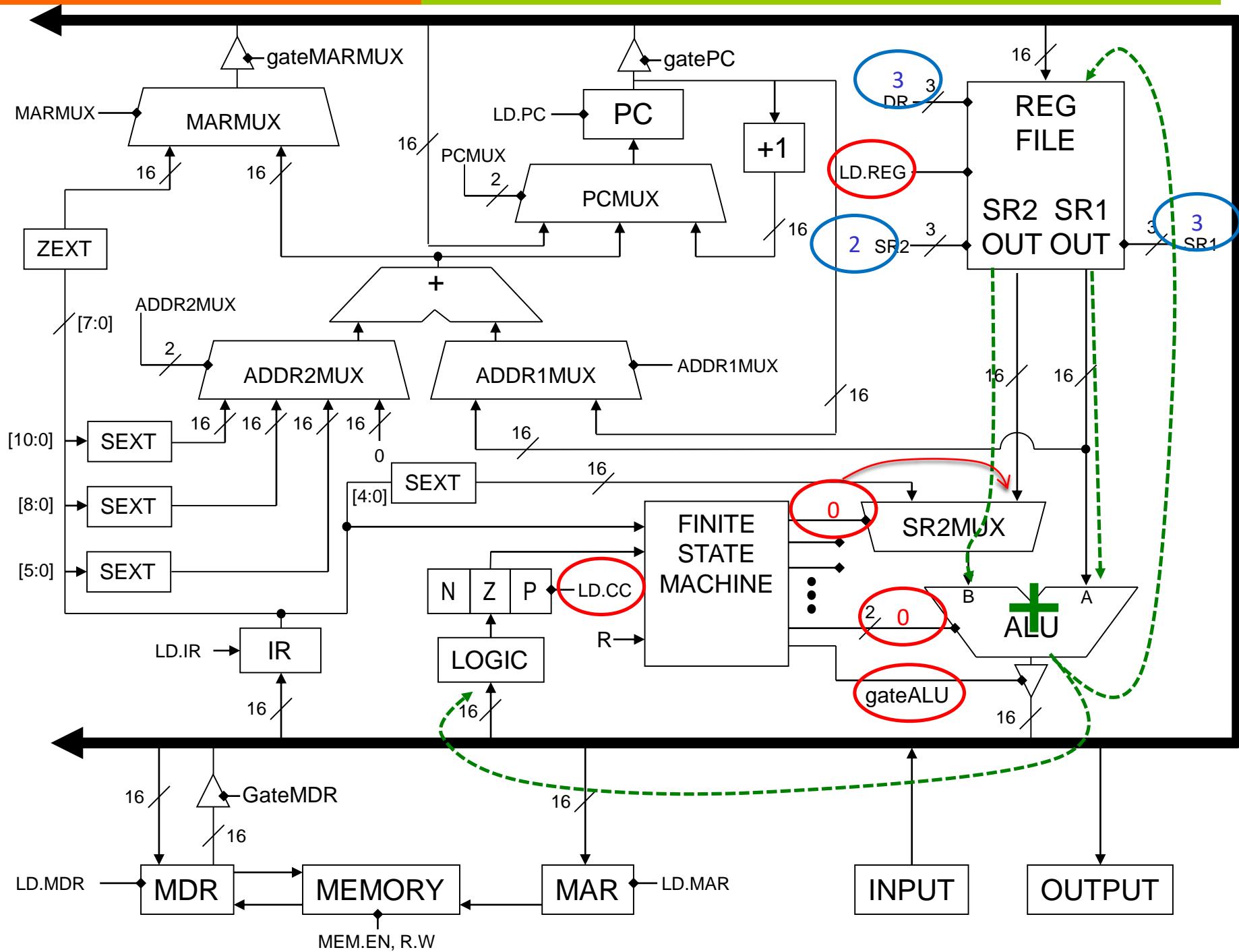
ADD R3, R3, #2

0001 011 011 1 00010

Literal or Immediate
Addressing Mode

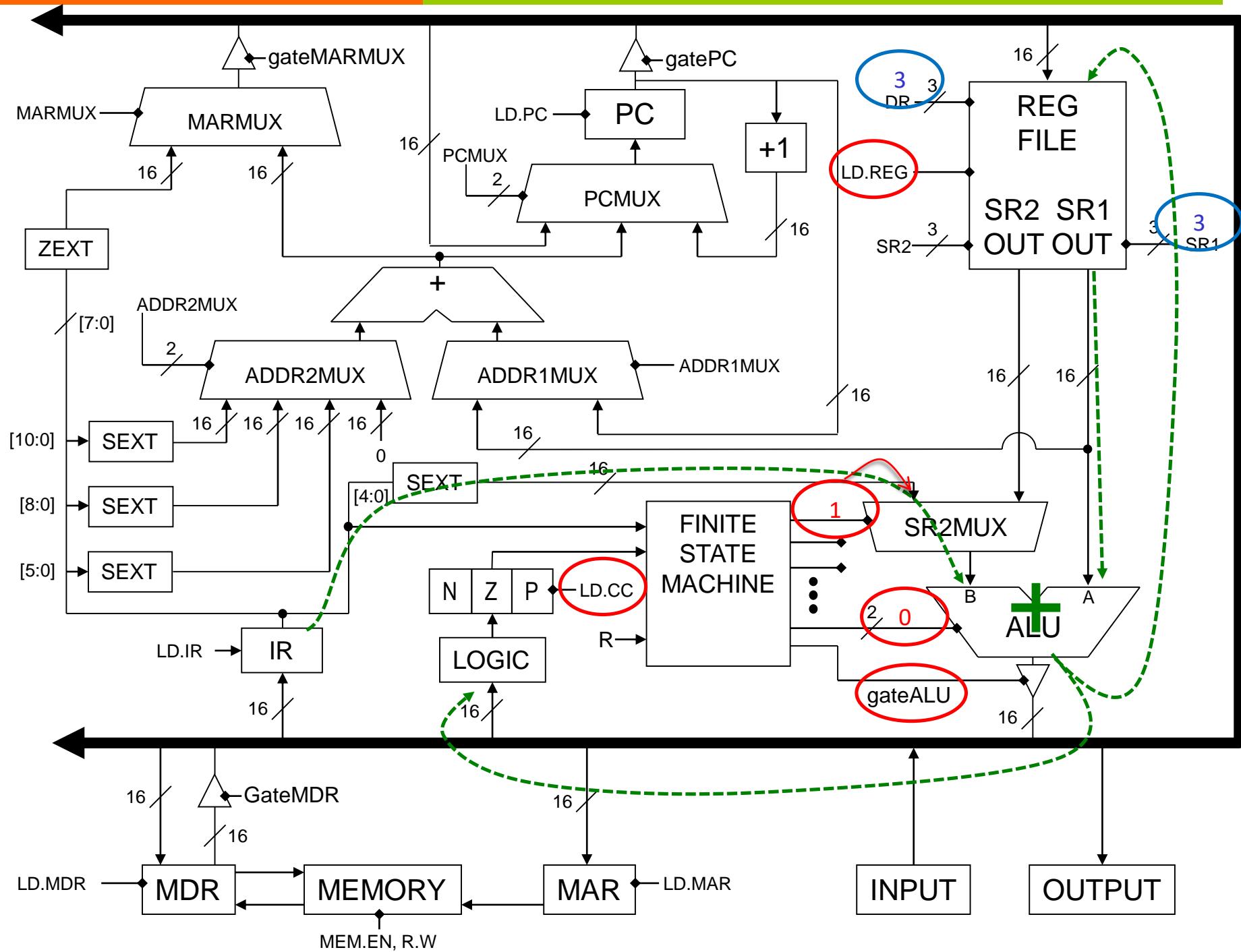
ADD R3, R3, R2 (Execute)

- Copy R3 to ALU Input A
- Copy R2 to ALU input B
- Set ALU to ADD
- Copy ALU output to R2
- And copy ALU output to CC



ADD R3, R3, #2 (Execute)

- Copy R3 to ALU Input A
- Sign-extend IR[4:0] to ALU input B
- Set ALU to ADD
- Copy ALU output to R3
- And copy ALU output to CC



Data Movement Instructions

Operate (ALU)

- ADD
- AND
- NOT

Data Movement (Memory)

Load

- LD
- LDR
- LDI
- LEA

Store

- ST
- STR
- STI

Control

- BR
- JMP
- JSR
- JSRR
- RET
- RTI
- TRAP

Data Movement Instructions

- Purpose:

- **Load** data into a register
 - Usually from memory
- **Store** a register's value out to memory

Addressing Modes

- ↗ Where can operands (data) be found?
 1. In the instruction (Literal or Immediate)
 2. In registers
 3. In memory
- ↗ We use the term ***Effective Address*** to describe the memory location that the instruction uses for its operands.

- ↗ How big is a machine instruction?
 - ↗ 16 bits
- ↗ How big is a memory address?
 - ↗ 16 bits
- ↗ Can we fit a full memory address in a machine instruction?
 - ↗ No, because we need some bits for the op code and other parts of the instruction.

Addressing Modes

- Immediate or Literal
- Register
- PC-relative *
 - PC-relative is an historical term; the mode acts like PC + Offset
- Base + Offset *
- Indirect *
 - * These 3 addressing modes involve memory locations, at the effective address.
- Each instruction allows a **subset** of these addressing modes (*a common approach in a RISC architecture*)

Addressing Modes for Data Movement

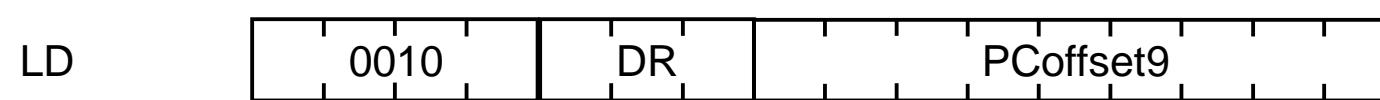
- Imagine memory is a large array:
 - `int mem[65536];`
- Calculate the effective address (EA), based on the addressing mode
 - PC relative:
 - `mem[PC + offset]`
 - Base + Offset:
 - `mem[BaseReg + offset]`
 - Indirect
 - `mem[mem[PC + offset]]`

Example: Calculate a PC-Relative Effective Address

- We can calculate a 16-bit memory address by adding an offset value to the 16-bit PC's (or general purpose register's) value
- For example, we can include a 9-bit offset value in the instruction itself (as two's complement)
 - Can represent values from -256 to +255
 - Let's call this value PCoffset9
- We can add PCoffset9 to the PC value (or a register value)
 - We must sign-extend PCOffset9 from 9 bits to 16 bits first.
- Because of the ISA design of the LC-3, we only discuss addressing modes and effective addresses for the **last** operand
 - Each operand but the last **always** uses Register addressing mode and the Effective Address is always the named register

Data Movement Instructions

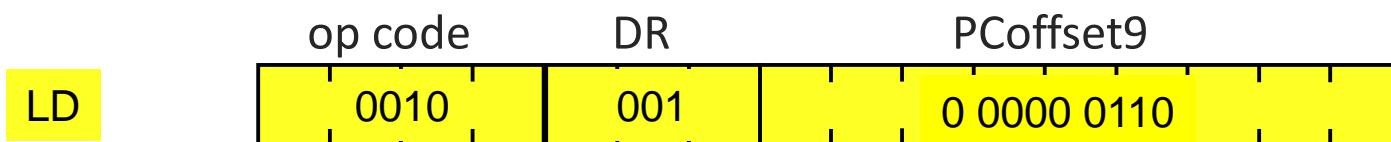
- Data movement instructions move
 - Memory to a register
 - A Register to memory
- Examples using PC-Relative addressing



Limitations?

PC relative addressing mode

<u>Address</u>	<u>Instruction</u>	
x3000	LD R1, #6	; PC == x3001 ; will load data from memory at EA, which is x3007 ; which is the value 3, and store 3 in R1
....		
x3007	x0003	; Some data, the number 3



PC: x3001

+ PCoffset9: x0006 (sign-extend this from 9 to 16 bits)

Effective Address (EA): x3007

Basic Format



These encode information on
how to form a 16 bit address
or a register address.

You'll see this described as
“OP2” in the LC-3 diagrams.

LD Example

<i>Addr</i>	<i>Content</i>	<i>Tag</i>	<i>Op</i>	<i>Operands</i>
			.orig	x3000
3000	0015		.fill	21
3001	27FE		LD	R3, #-2

0010	011	1	1111	1110
LD	R3		-2	

A red arrow points from the circled '-2' in the second row to the question below.

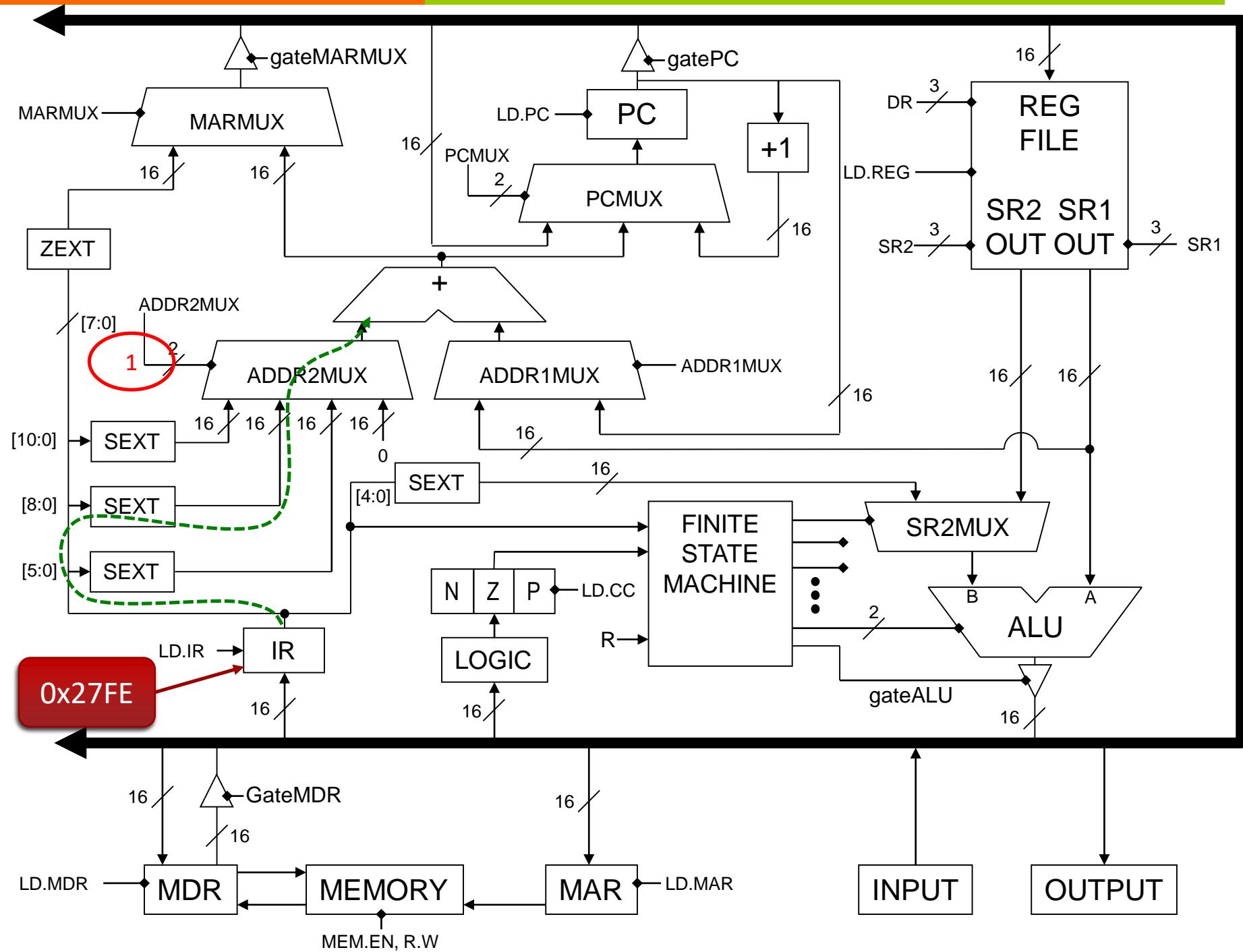
What happens when we execute at x3001?

Load from Memory: 3+ cycles

- 
1. Send the address to MAR
 2. Read memory[MAR] into MDR
 3. Send MDR back to a register
-
- ↗ Recall FETCH (get a machine code instruction from memory)
 - ↗ Load (LD) is similar – but its execution state gets data from memory into a register

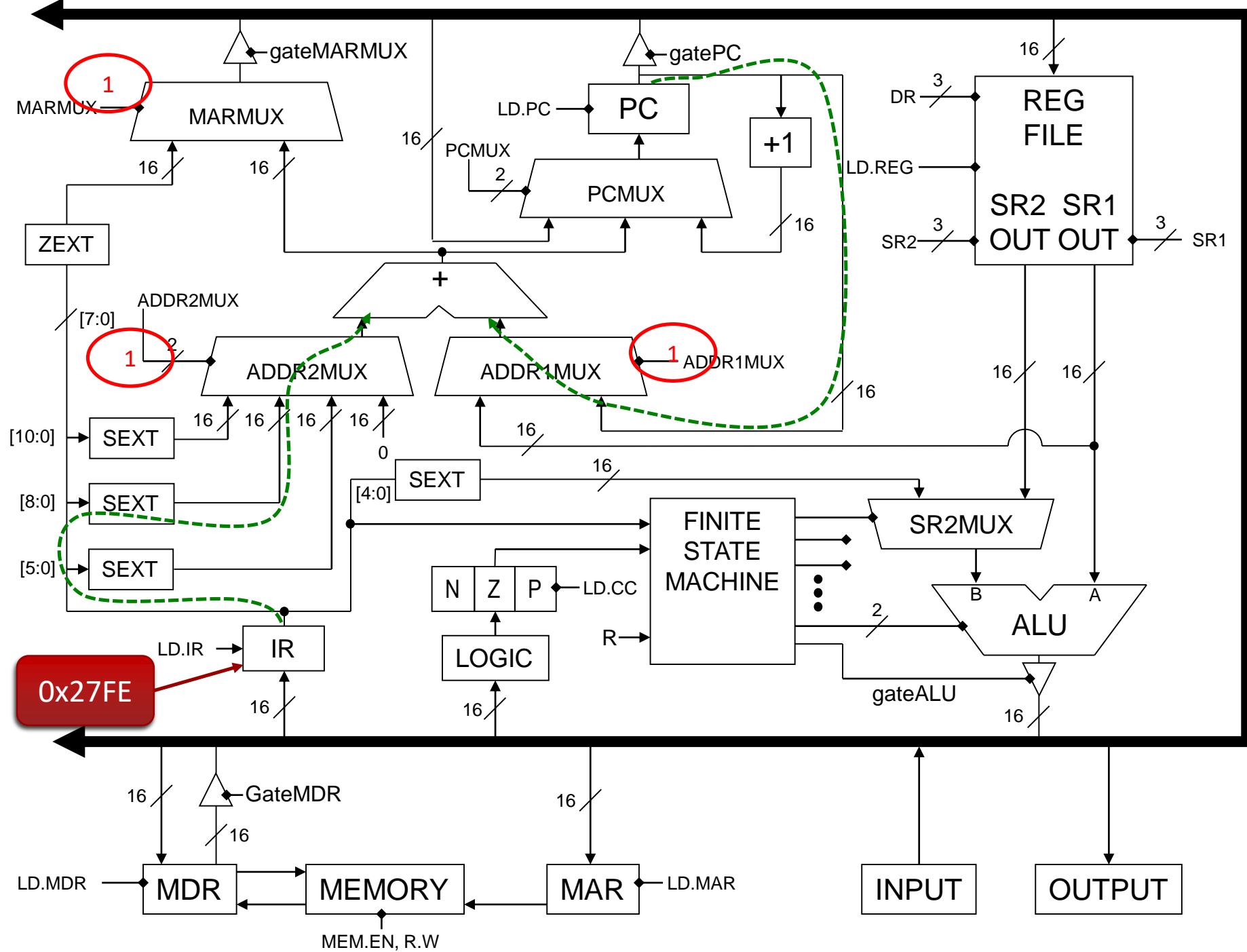
LD R3,-2 (Execute state 1)

- Add SEXT(IR[8:0]) to the (incremented) PC and store in MAR



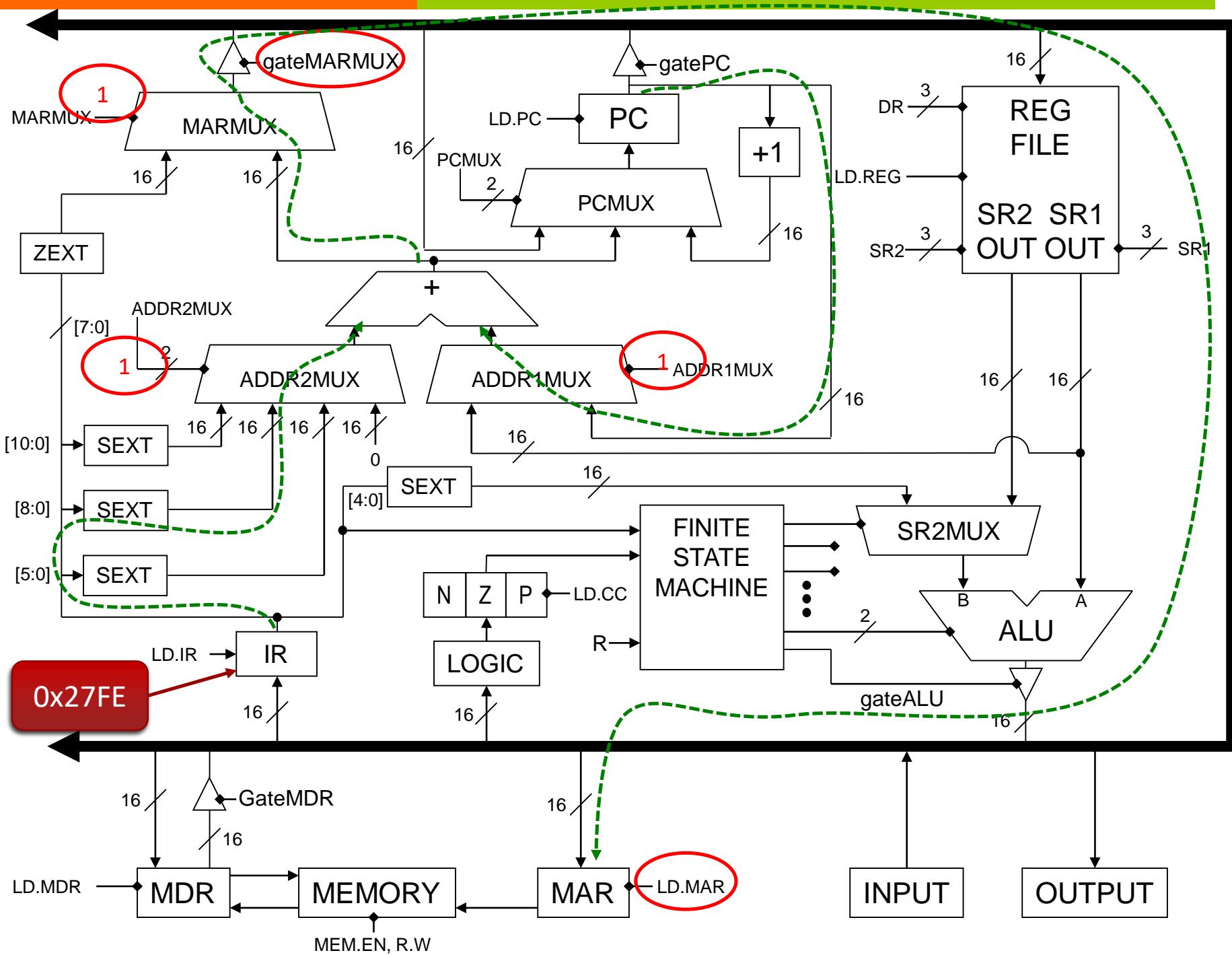
LD R3,-2 (Execute state 1)

- Add SEXT(IR[8:0]) to the (incremented) PC and store in MAR



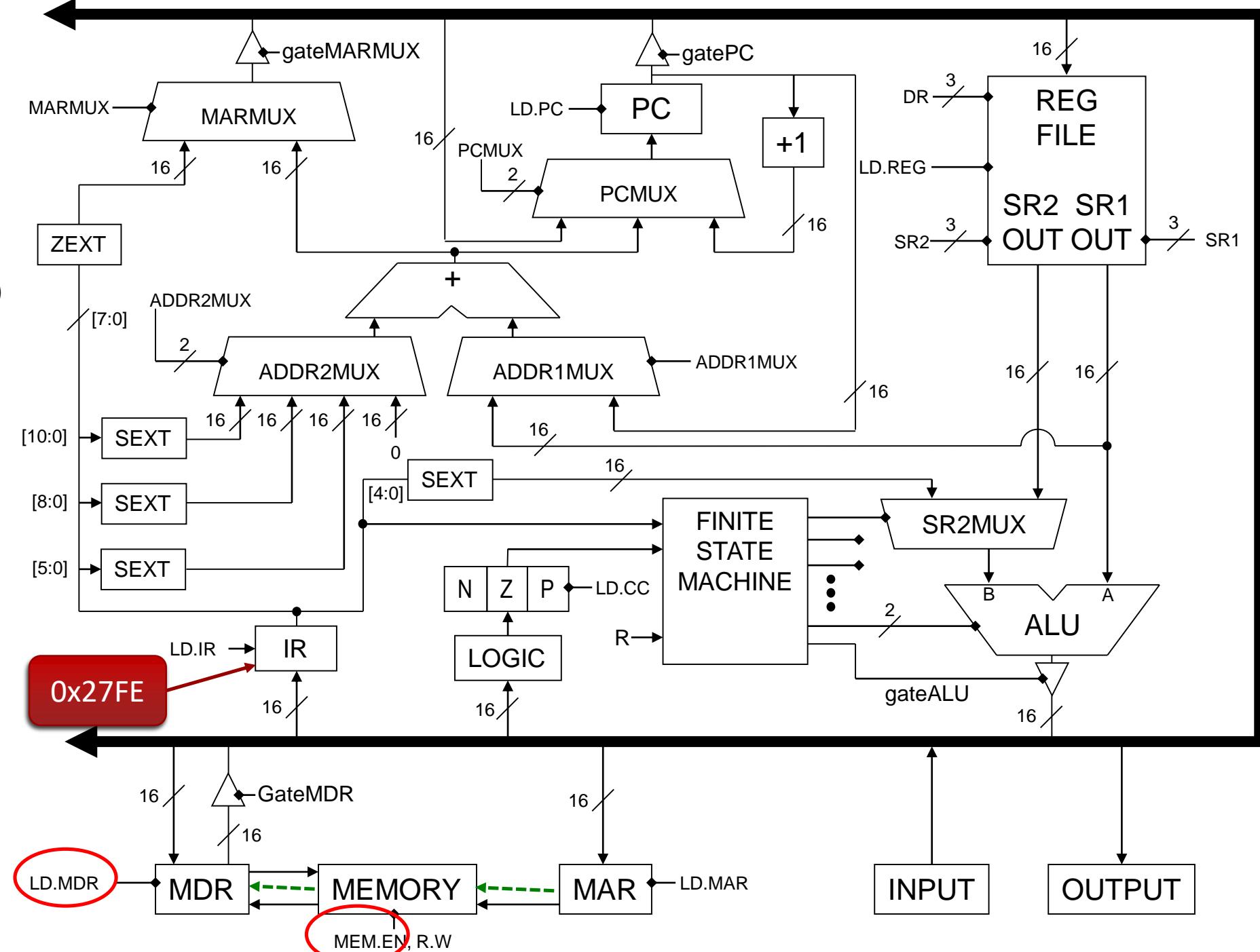
LD R3,-2 (Execute state 1)

- Add SEXT(IR[8:0]) to the (incremented) PC and store in MAR



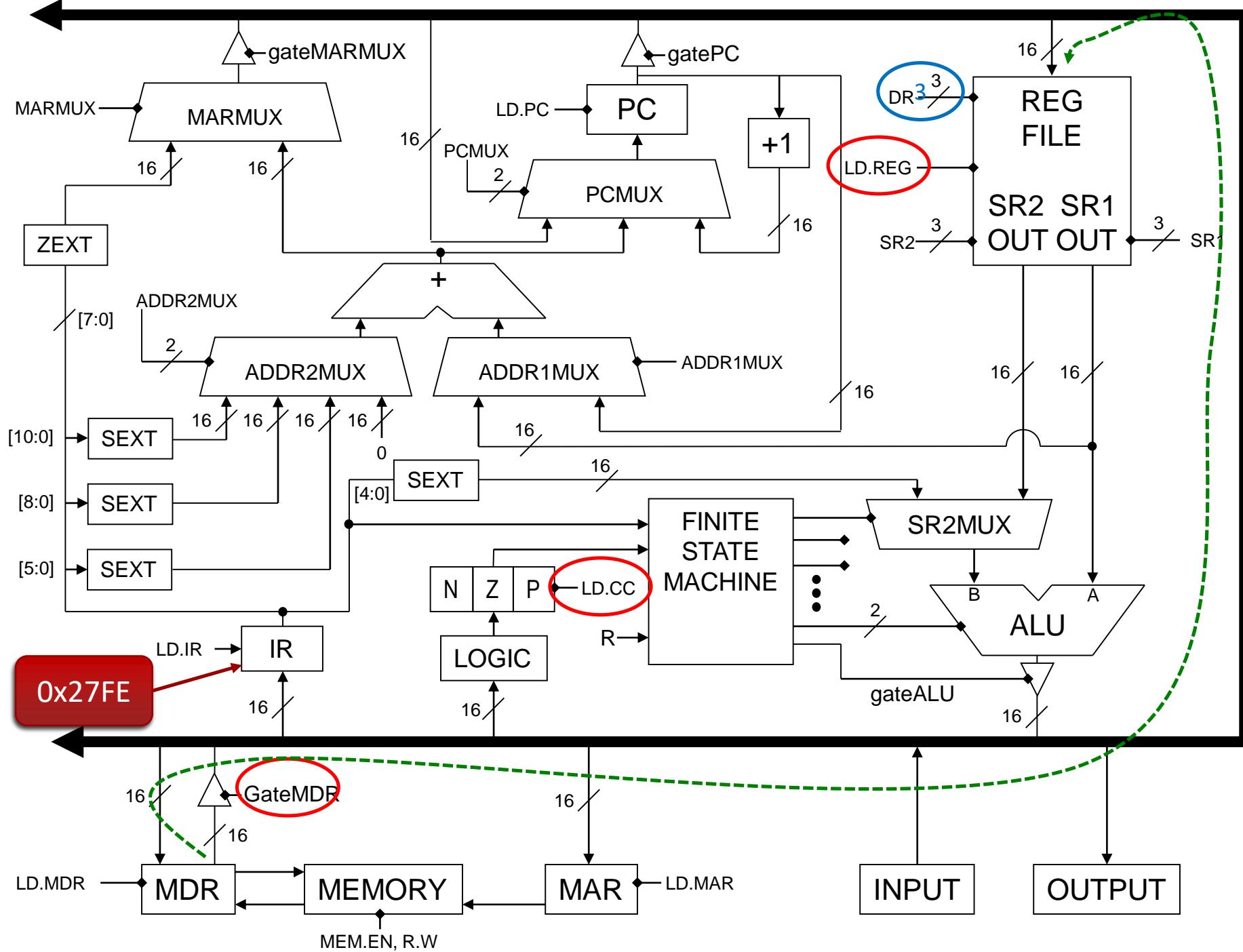
LD R3,-2 (Execute state 2)

- Add SEXT(IR[8:0]) to the (incremented) PC and store in MAR
- Raise MEM.EN and LD.MDR to cause a memory fetch from mem[MAR] into MDR



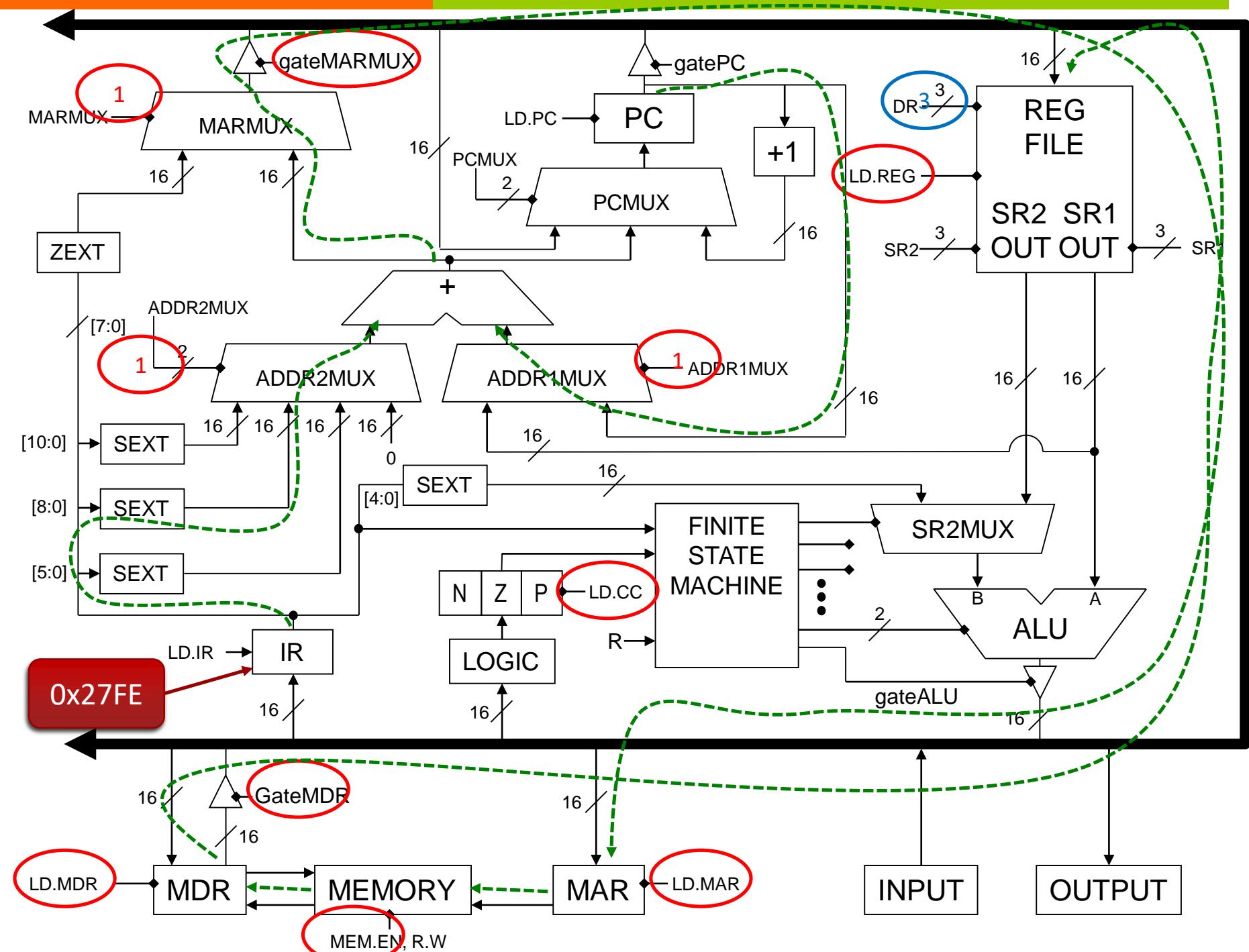
LD R3,-2 (Execute state 3)

- Add SEXT(IR[8:0]) to the (incremented) PC and store in MAR
- Raise MEM.EN and LD.MDR to cause a memory fetch from mem[MAR] into MDR
- Copy the value in MDR into R3, setting the condition codes



LD R3,-2 (Signals)

- ADDR2MUX=1
ADDR1MUX=1
MARMUX=1
gateMARMUX
LD.MAR
- MEM.EN
LD.MDR
- gateMDR
DR=IR[11:9]
LD.REG
LD.CC



Data Movement Instructions

↗ Base + Offset

LDR+



STR



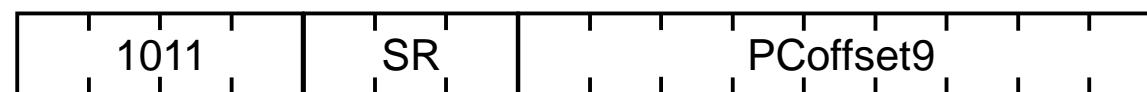
Data Movement Instructions

↗ Indirect Mode

LDI+



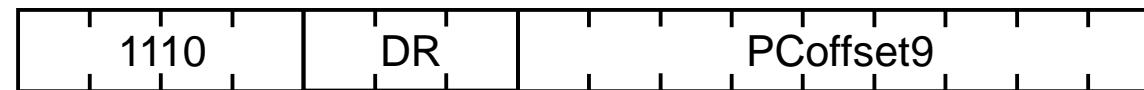
STI



Data Movement Instructions

↗ Immediate Mode

LEA



No memory access!

Where's the Operand?

Addressing Mode	Operand
Immediate	SEXT(IR[4:0])
Register	Use the contents of the specified register
PC-relative	Compute PC + SEXT(IR[8:0]) or SEXT(IR[10:0]) and dereference -- remember that PC is already incremented by this time in the instruction cycle
Base + Offset	Compute specified register + SEXT(IR[5:0]) and dereference
Indirect	Compute PC-relative address as above in PC-relative, but dereference* twice

* Dereference : Interpret what you have as a memory address and fetch

Which instructions use the Base + Offset addressing mode?

- A. LDR, STR
- B. JMP
- C. BRP
- D. LDI, STI

Compare and Contrast

- ↗ LEA DR \leftarrow PC + PCOffset9
- ↗ LD DR \leftarrow Mem [PC + PCOffset9]
- ↗ LDI DR \leftarrow Mem [Mem [PC + PCOffset9]]
- ↗ LDR DR \leftarrow Mem [BaseR + Offset6]

- ↗ LEA: Puts the effective address (PC+offset) itself into a register (*no memory access!*)
- ↗ LD: Puts the contents of some memory at the effective address (PC+offset) into a register
- ↗ LDR: Puts the contents of the effective address (base register+offset) into a register.
- ↗ LDI: Goes to the effective address (PC+offset) and gets a value. Treats that value as another memory address, and gets data from there, into a register (*hits memory twice*)

Store to Memory: 3+ cycles

1. Send the address to MAR
 2. Send the data to MDR
 3. Store MDR into memory[MAR]
- ☞ NOTE: Indirect addressing mode takes 5 cycles
- ☞ STI (store indirect)
 - ☞ LDI (load indirect) also takes 5 cycles

- ↗ ST: Puts the contents of a register into memory at the effective address (PC+offset).
- ↗ STR: Puts the contents of a register into memory at the effective address (base register+offset)
- ↗ STI: Puts the contents of a register into a memory location whose address is stored in memory at the effective address (PC+offset).
(Hits memory twice!)

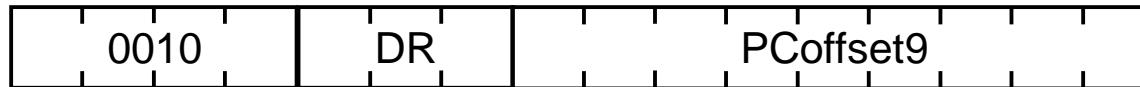
A word of advice

- Probably the most confusing thing for the new assembly language programmer is the confusion between memory addresses and memory contents
- Consider

```
int x = 10;
x = x + 2;
```
- What exactly do we mean by **x**?
- **x** is a name given to a memory address, but it means something different depending on which side of the assignment it appears
- On the left, it is a place in which to store the assigned value; on the right it means “the value stored at memory address **x**”
- Programming languages do this implicit dereferencing of the right side variables for you; assembly language does not!

LD Example

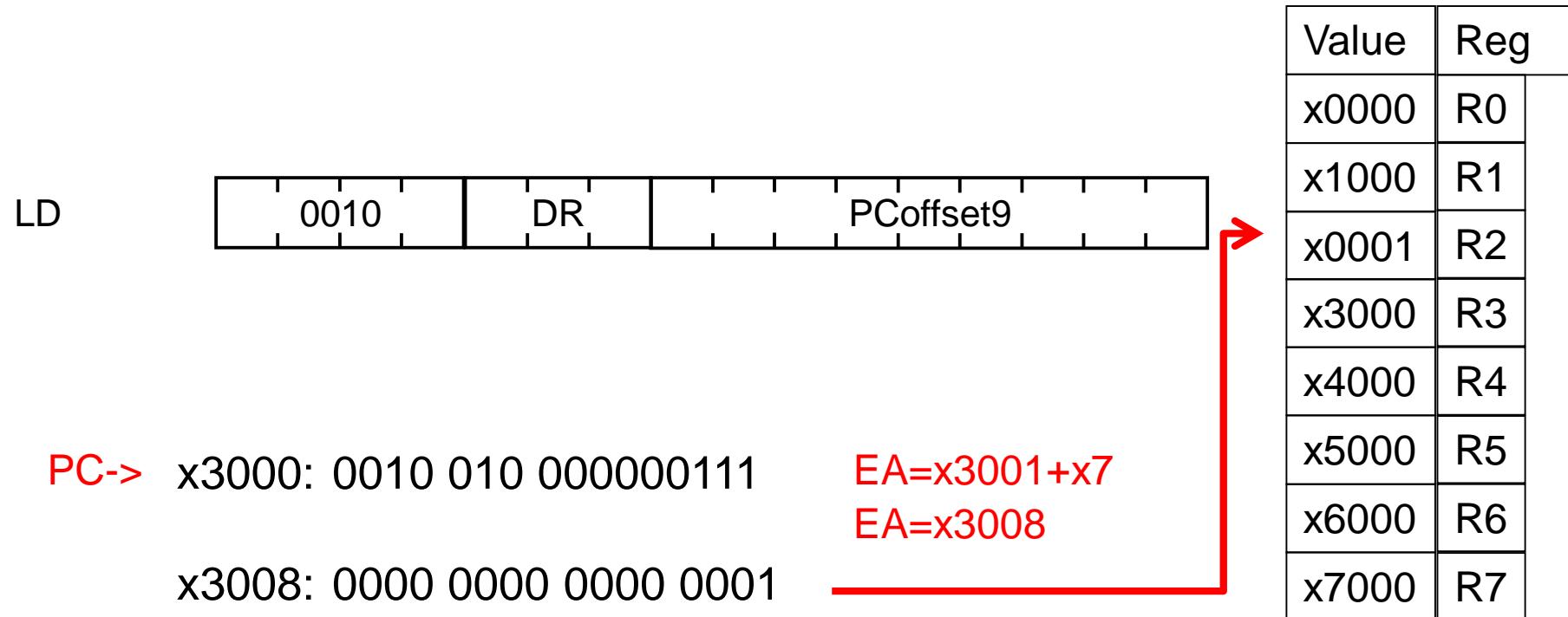
LD



PC-> x3000: 0010 010 000000111 EA=x3001+x7
 EA=x3008
 x3008: 0000 0000 0000 0001

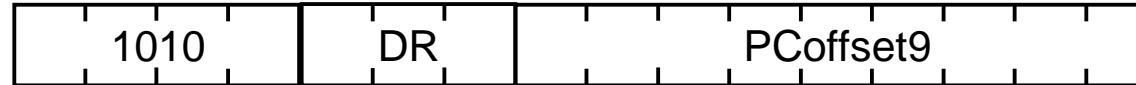
Value	Reg
x0000	R0
x1000	R1
x2000	R2
x3000	R3
x4000	R4
x5000	R5
x6000	R6
x7000	R7

Example



LDI Example

LDI



PC-> x3000: 1010 011 000000111 EA=M[x3001+x7]

EA=M[x3008]

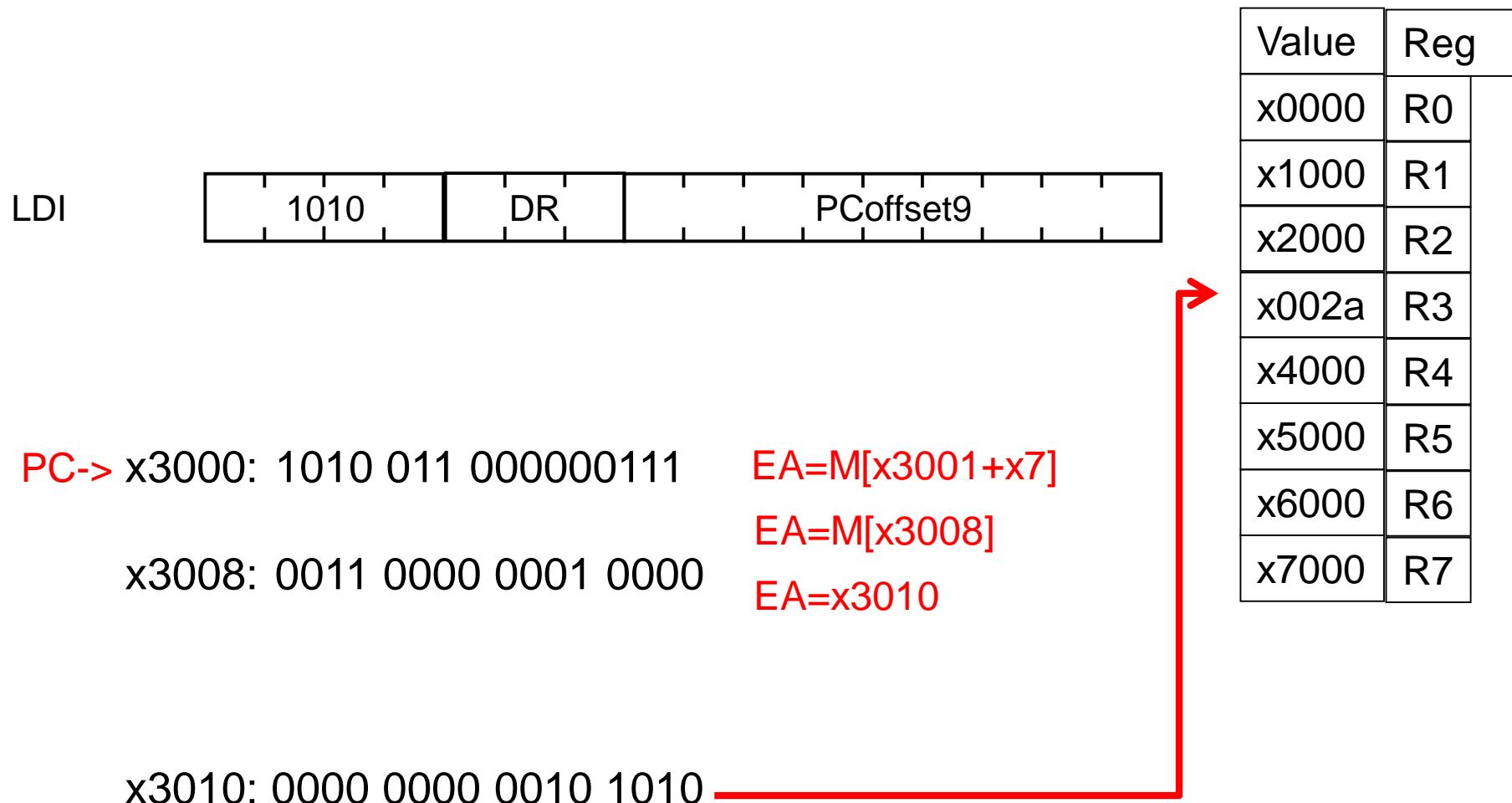
x3008: 0011 0000 0001 0000

EA=x3010

x3010: 0000 0000 0010 1010

Value	Reg
x0000	R0
x1000	R1
x2000	R2
x3000	R3
x4000	R4
x5000	R5
x6000	R6
x7000	R7

LDI Example



STR Example

STR



PC->x3000: 0111 110 011 000011 EA=X3010+x3
EA=X3013

Value	Reg
x0000	R0
x1000	R1
x2000	R2
x3010	R3
x4000	R4
x5000	R5
x6000	R6
x7000	R7

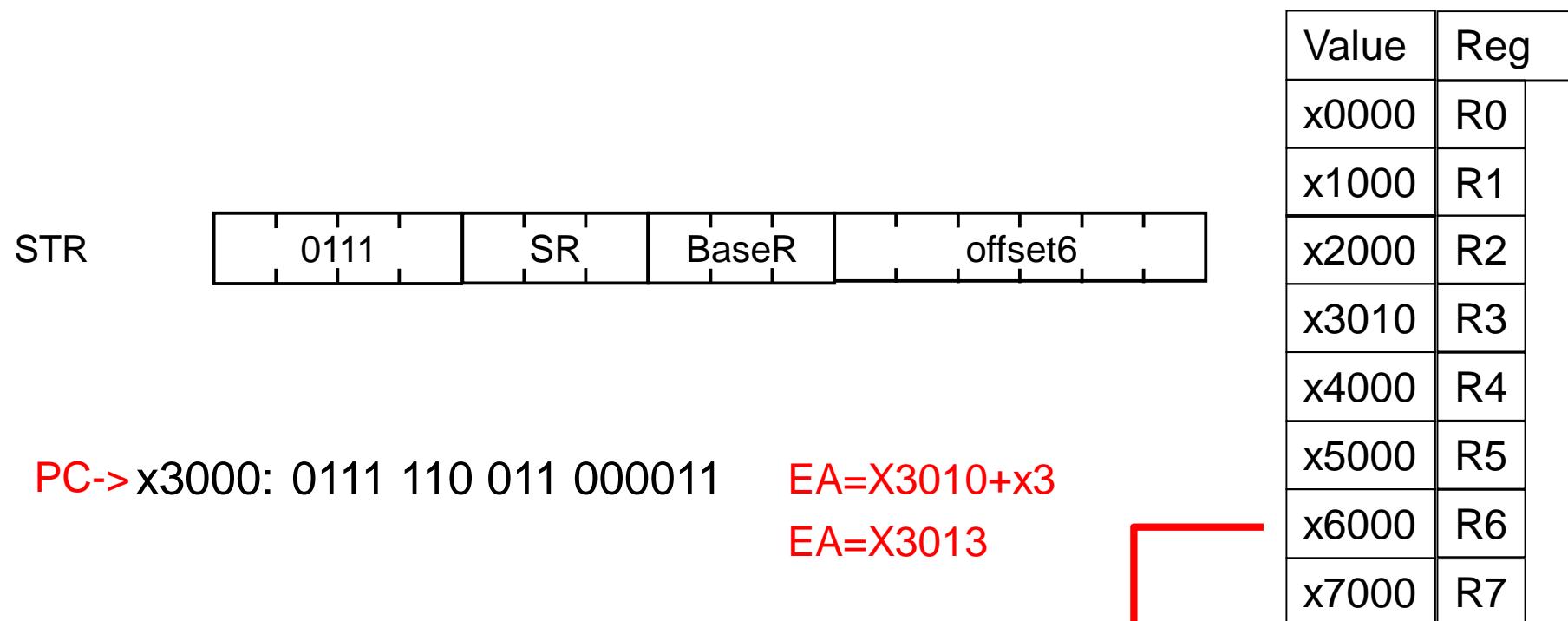
x3010: 0000 0000 0010 1010

x3011: 0000 0000 0000 0000

x3012: 0000 0000 0000 1111

x3013: 0000 0000 1111 0000

STR Example



x3010: 0000 0000 0010 1010

x3011: 0000 0000 0000 0000

x3012: 0000 0000 0000 1111

x3013: 0110 0000 0000 0000

LEA Example

LEA



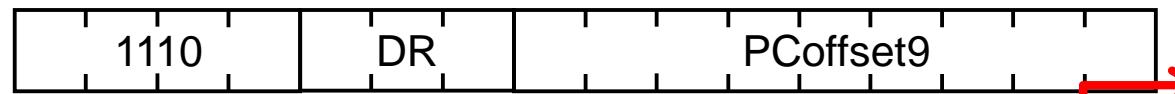
PC-> x3000: 1110 011 0 0000 1111 EA=x3001+xF
EA=x3010

Value	Reg
x0000	R0
x1000	R1
x2000	R2
x3000	R3
x4000	R4
x5000	R5
x6000	R6
x7000	R7

x3010: 0000 0000 0010 1010
x3011: 0000 0000 0000 0000
x3012: 0000 0000 0000 1111
x3013: 0000 0000 1111 0000

LEA Example

LEA



PC-> x3000: 1110 011 0 0000 1111 EA=x3001+xF
EA=x3010

Value	Reg
x0000	R0
x1000	R1
x2000	R2
x3010	R3
x4000	R4
x5000	R5
x6000	R6
x7000	R7

x3010: 0000 0000 0010 1010
x3011: 0000 0000 0000 0000
x3012: 0000 0000 0000 1111
x3013: 0000 0000 1111 0000

Example – Assembly Code Tracing

Address	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x30F6	1	1	1	0	0	0	1	1	1	1	1	1	1	0	1	R1<- PC-3
x30F7	0	0	0	1	0	1	0	0	0	1	1	0	1	1	1	R2<- R1+14
x30F8	0	0	1	1	0	1	0	1	1	1	1	1	1	0	1	M[x30F4] <- R2
x30F9	0	1	0	1	0	1	0	0	1	0	1	0	0	0	0	R2<- 0
x30FA	0	0	0	1	0	1	0	0	1	0	1	0	0	1	0	R2<- R2+5
x30FB	0	1	1	1	0	1	0	0	0	1	0	0	1	1	1	M[R1+14] <- R2
x30FC	1	0	1	0	0	1	1	1	1	1	1	0	1	1	1	R3<- M[M[x3F04]]

Opcodes

ADD	0x1
LEA	0xE
ST	0x3
AND	0x5
STR	0x7
LDI	0xA

		R1	R2	R3	x30f4	x3102
LEA	R1, #-3	x30f4	-	-	-	-
ADD	R2, R1, #14		x3102			
ST	R2, #-5				x3102	
AND	R2, R2, #0		0			
ADD	R2, R2, #5		5			
STR	R2, R1, 14					5
LDI	R3, #-9			5		

Control Instructions

Operate (ALU)

- ADD
- AND
- NOT

Data Movement (Memory)

- Load
- LD
 - LDR
 - LDI
 - LEA

- Store
- ST
 - STR
 - STI

Control

- BR
- JMP
- JSR
- JSRR
- RET
- RTI
- TRAP

Question

If you don't want program execution to proceed to the next instruction in sequence, what register must be changed?

- a) Instruction Register (IR)
- b) Memory Address Register (MAR)
- c) Memory Data Register (MDR)
- d) Program Counter (PC) 

Changing the Sequence of Instructions

- ↗ In the FETCH phase, we increment the Program Counter by 1.
- ↗ What if we don't want to execute the instruction that follows this one?
 - ↗ examples: loop, if-then, function call
- ↗ We need special instructions that change the contents of the PC.
- ↗ These are called *control instructions*.
 - ↗ **jumps** are unconditional -- they always change the PC
 - ↗ **branches** are conditional -- they change the PC only if some condition is true (e.g., the result of an ADD is zero)

Example: LC-3 JMP Instruction

- Set the PC to the value contained in a register. This becomes the address of the next instruction to fetch.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	0	0	0	Base	0	0	0	0	0	0	0	0	0	0	0

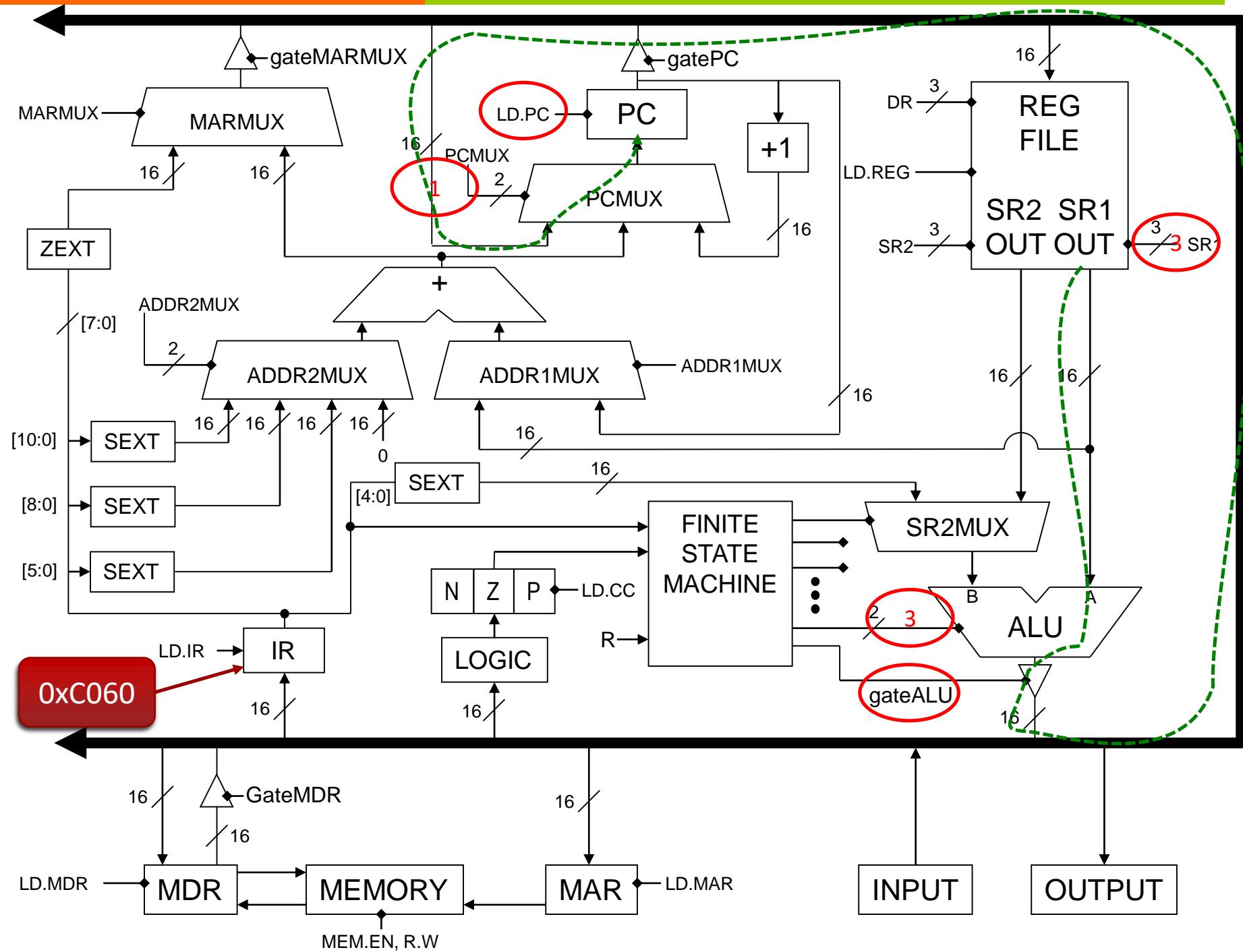
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	0	0	0	0	0	0	1	1	0	0	0	0	0	0

“Move the contents of R3 into the PC.”

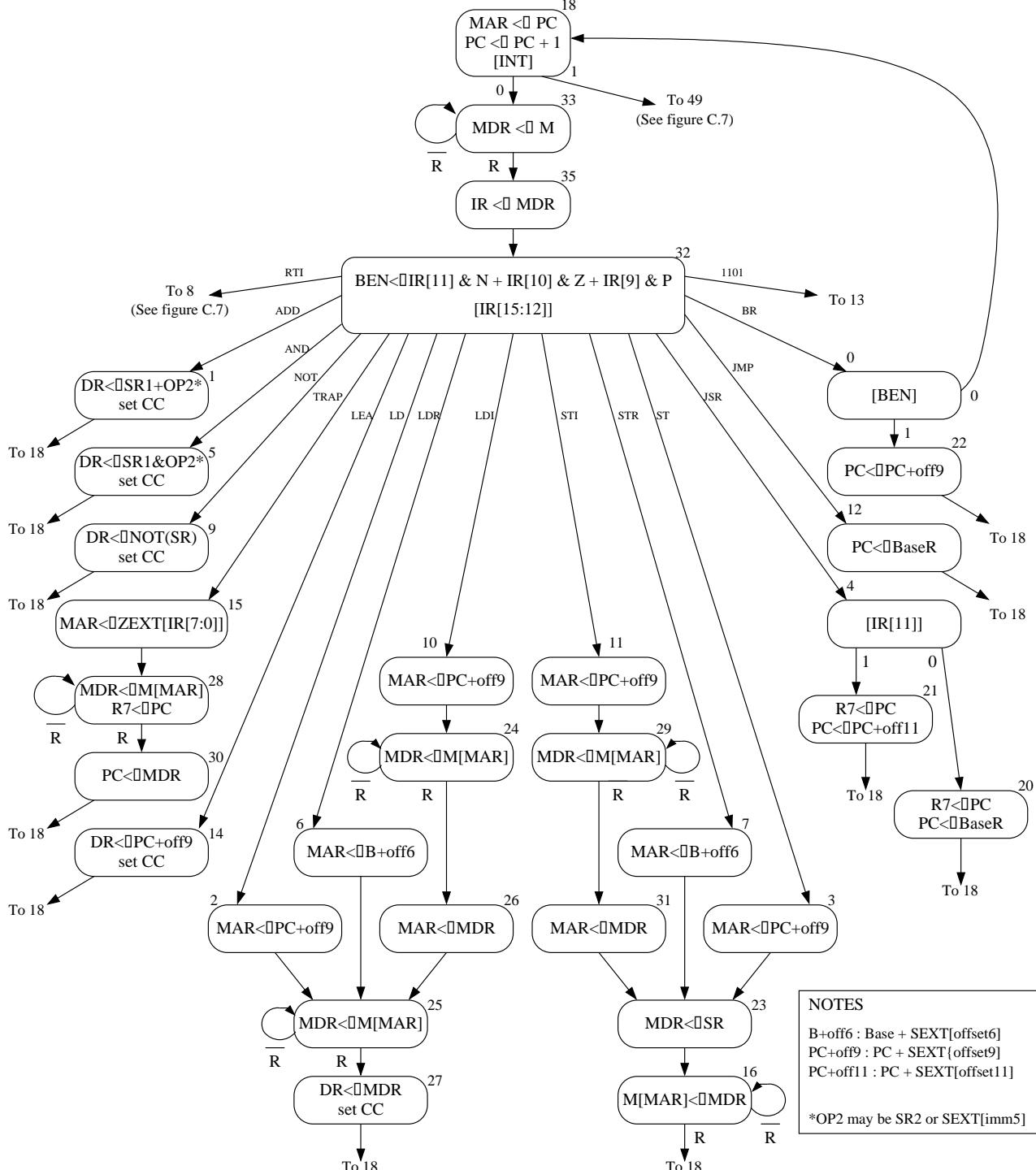
JMP R3 (Execute state)

Copy R3 through
ALU, bus, and
PCMUX to PC

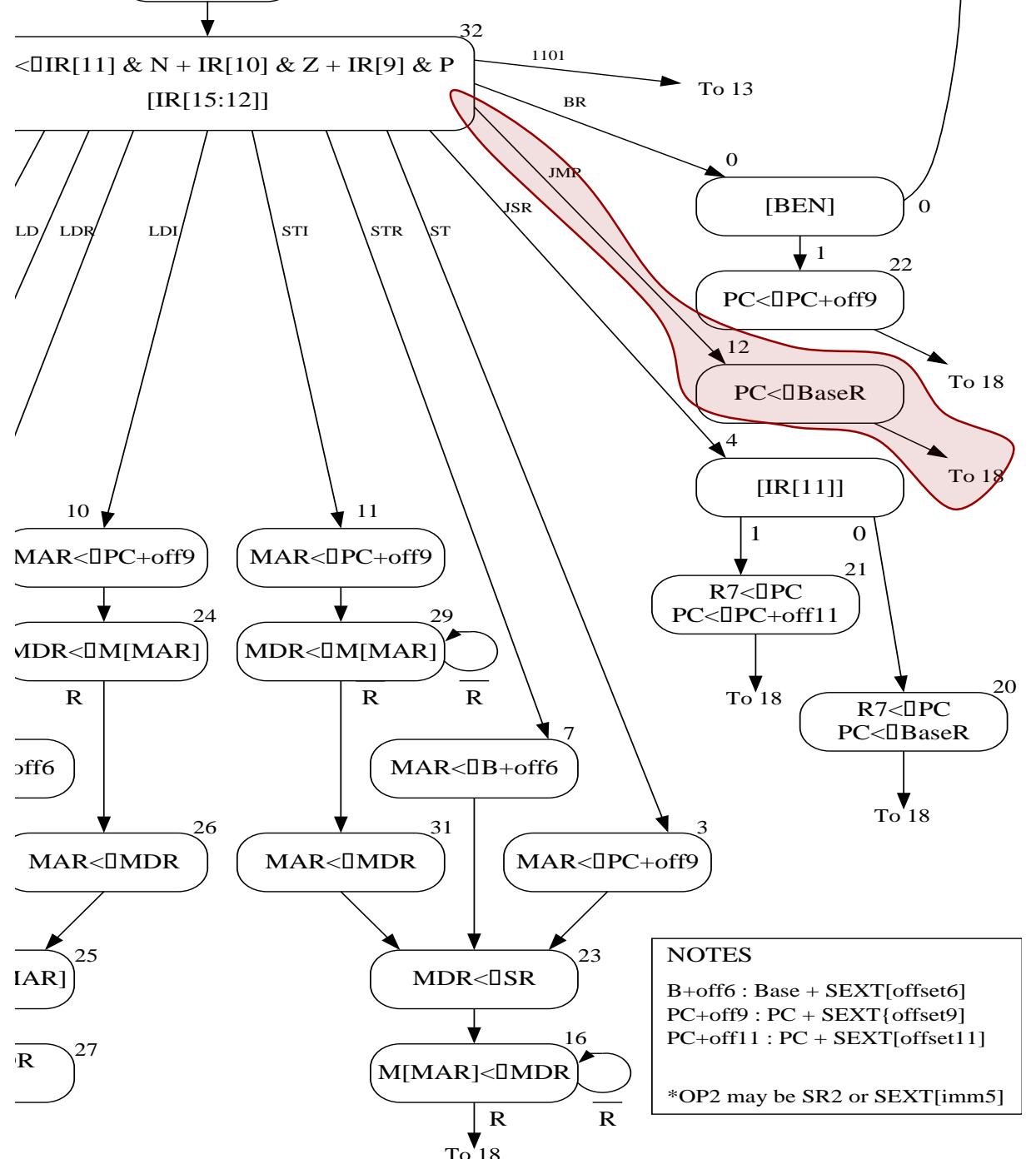
SR1=IR[8:6]
SR1MUX=1
ALUK=3 (PASS A)
gateALU
PCMUX=1
LD.PC



LC-3 FSM State Diagram (Part 2ed)



FSM States for Executing JMP

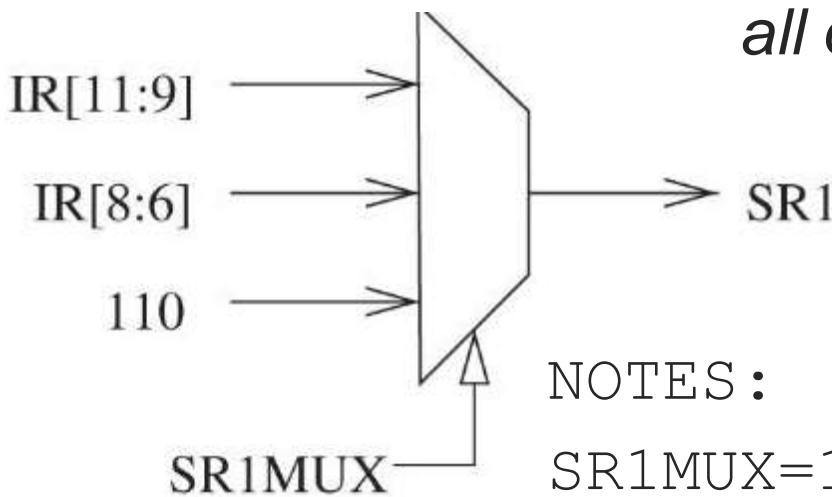


EXECUTE State for JMP

001100 (State 12) 0 0 J=18

LD.PC GateALU PCMUX=1 SR1MUX=1 ALUK=3

all other control signals are 0



NOTES:
SR1MUX=1 means use IR[8:6] as SR1

ALUK=3 means "Pass A"

LC-3 Instructions, first half

ADD+	0001	DR	SR1	0	00	SR2
ADD+	0001	DR	SR1	1		imm5
AND+	0101	DR	SR1	0	00	SR2
AND+	0101	DR	SR1	1		imm5
BR	0000	n	z	p		PCoffset9
JMP	1100	000	BaseR		000000	
JSR	0100	1			PCoffset11	
JSRR	0100	0	00	BaseR		000000
LD+	0010	DR			PCoffset9	
LDI+	1010	DR			PCoffset9	

+ Indicates instructions that modify condition codes

LC-3 Instructions, second half

LDR+	0110	DR	BaseR	offset6
LEA	1110	DR		PCoffset9
NOT+	1001	DR	SR	111111
RET	1100	000	111	000000
RTI	1000		000000000000	
ST	0011	SR		PCoffset9
STI	1011	SR		PCoffset9
STR	0111	SR	BaseR	offset6
TRAP	1111	0000		trapvect8
reserved	1101			

+ Indicates instructions that modify condition codes

Control Instructions

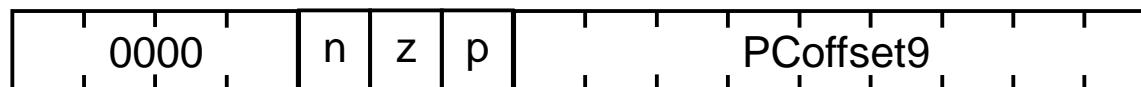
JMP




Set PC to contents of BaseR

register

BR

Add offset to PC if condition matches

PC relative

JSR



JSRR

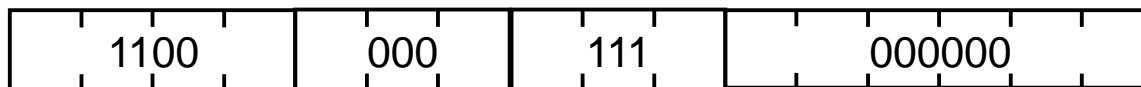



Save PC in R7, then add PCoffset11 or set to BaseR

PC relative

register

RET



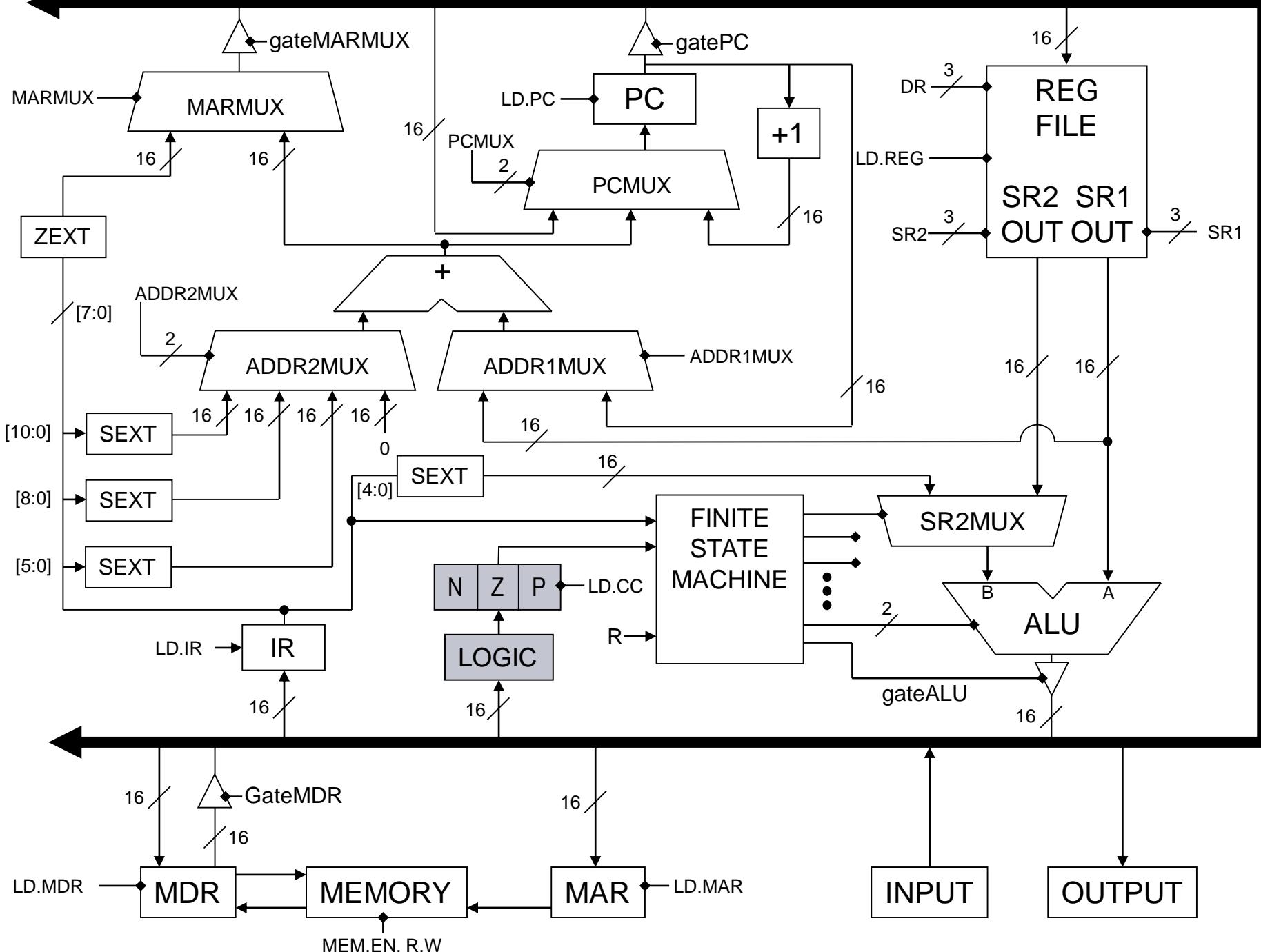
Set PC to contents of R7 (a.k.a. JMP R7)

register

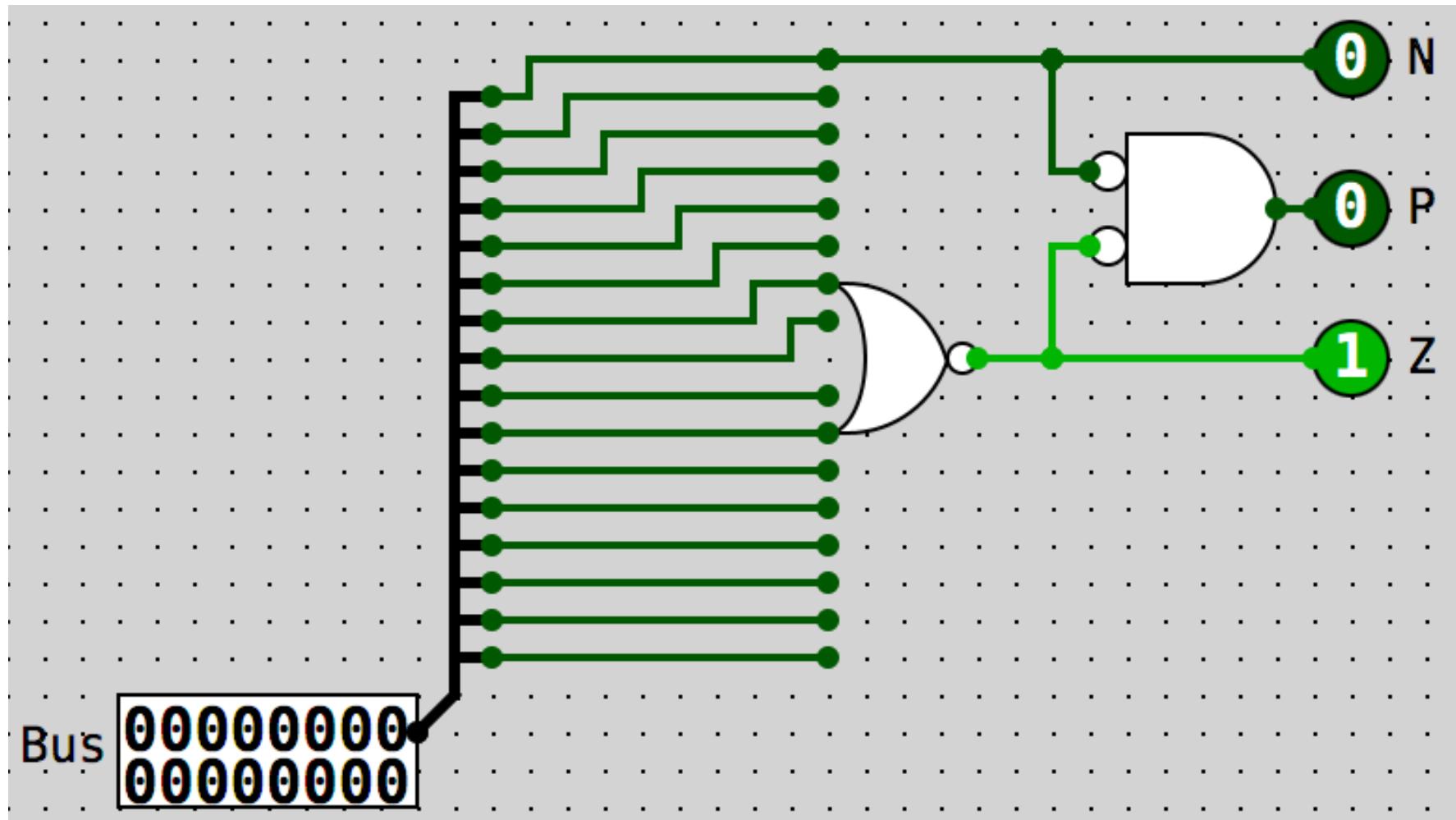
- ↗ N - Negative
- ↗ Z - Zero
- ↗ P - Positive

- ↗ When is the condition code register set?
 - ↗ During EXECUTE when LD.CC is high
 - ↗ Only set by ADD, AND, NOT, LD, LDR, LDI
 - ↗ Otherwise the CCs remain unchanged
- ↗ How are they used?
 - ↗ There are 3 bits in the BR instruction that are ANDed with the NZP condition code bits. If any bits are on, the branch is taken; otherwise the instruction in the following word is fetched next

Branch Logic



Condition Code Logic



Which instructions set Condition Codes?

- All the ALU instructions
 - ADD, AND, NOT
- the LOAD (data movement) instructions
 - LD, LDR, LDI
- In other words, every instruction that changes the value of a general purpose register, R0-R7
 - Except for LEA
- All other instructions do NOT change the condition code.

Which of these groups of instructions all set the condition codes (NZP)?

- A. LEA, LD, LDR
- B. ST, STR, STI
- C. JMP, JSRR, RET
- D. LD, LDR, ADD, NOT 

Questions?

What registers are transferred during the EXECUTE phase of
JMP R3

?

- A. PC <- R7
- B. R7 <- PC
- C. PC <- R3
- D. R7 <- PC
PC <- R3



Today's number is 21,021

DECODE, in the datapath

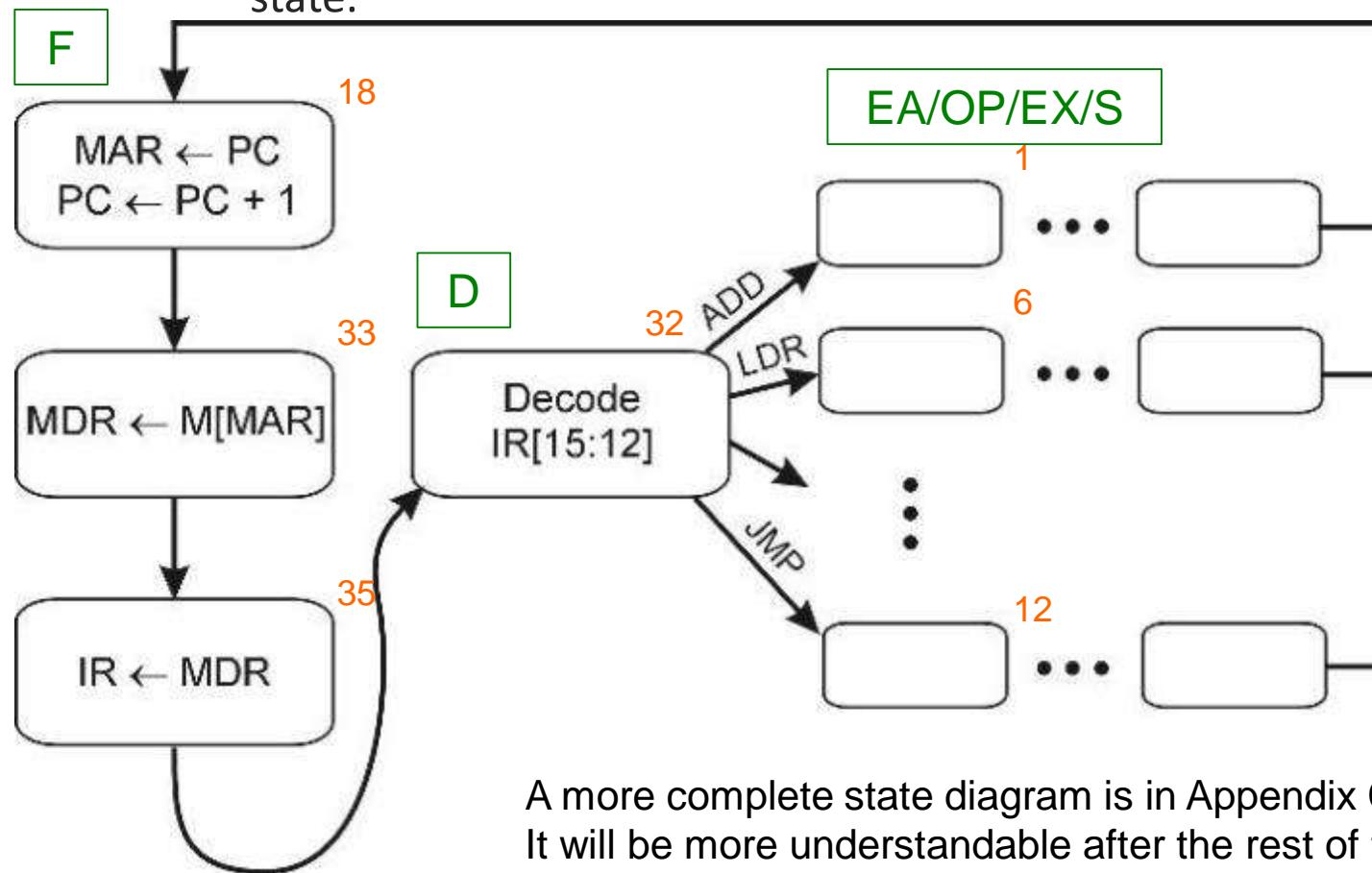
- Recall, every instruction starts with these four states:
 - Fetch
 - Fetch
 - Fetch
 - Decode
- How does DECODE work?
 - This is where the state machine is implemented in the LC-3

How Does that Control Unit (FSM) Work?

- The control unit controls the flow of data in the LC-3 data path
- Rather than have a big sequential logic circuit, the Control Unit executes a built-in program which is called ***microcode***
- Each cycle through the microcode (program) executes one LC-3 machine instruction
- The microcode itself runs on a smaller sequential logic circuit to carefully choose a sequence from 64 possible states
- The components of the control unit are
 - The ***Control Store***
 - The ***Microsequencer***

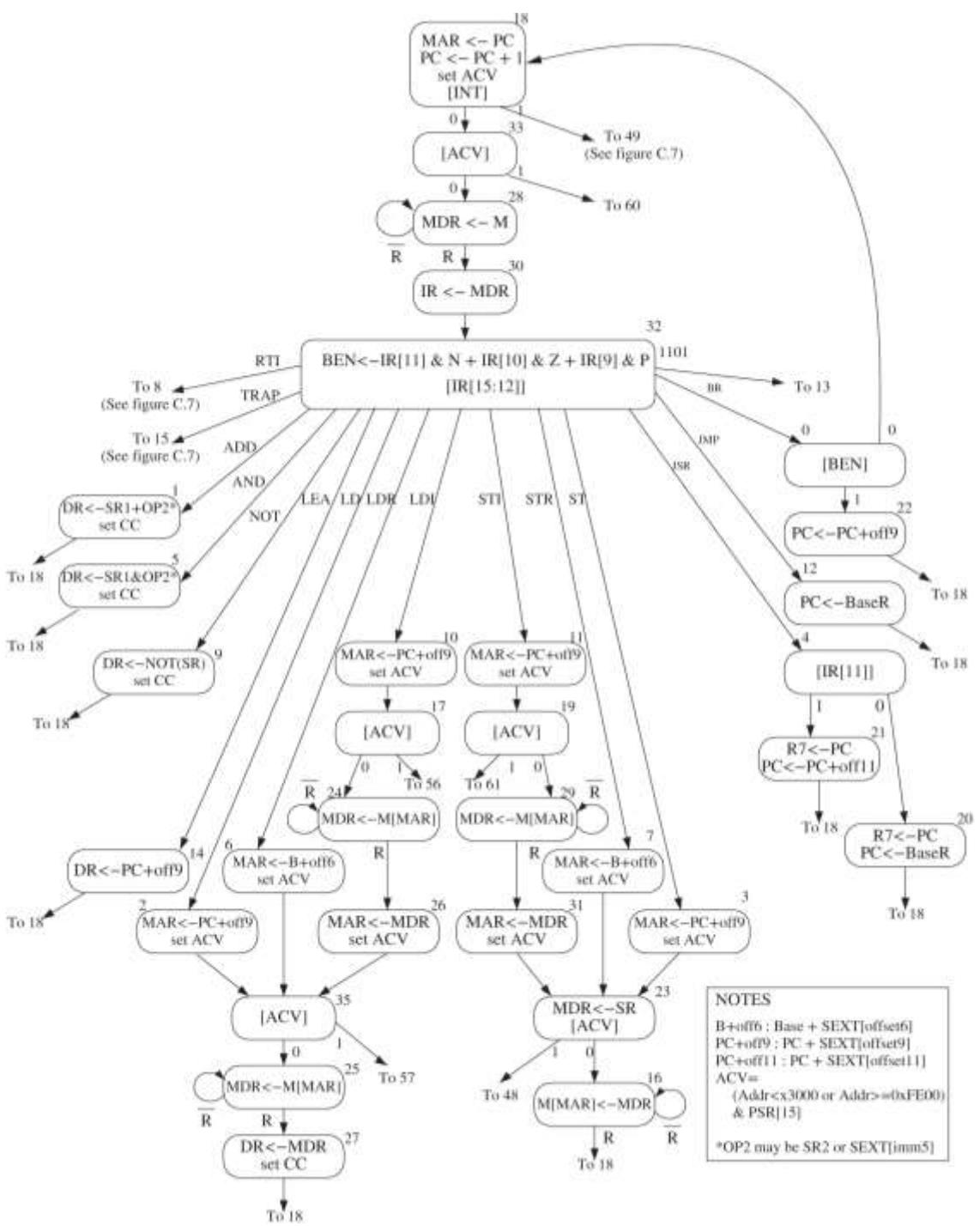
Recall LC-3 State Machine

- The FSM, which sequences through the states.
- It turns on the appropriate control signals as it enters each state.



A more complete state diagram is in Appendix C.
It will be more understandable after the rest of this deck

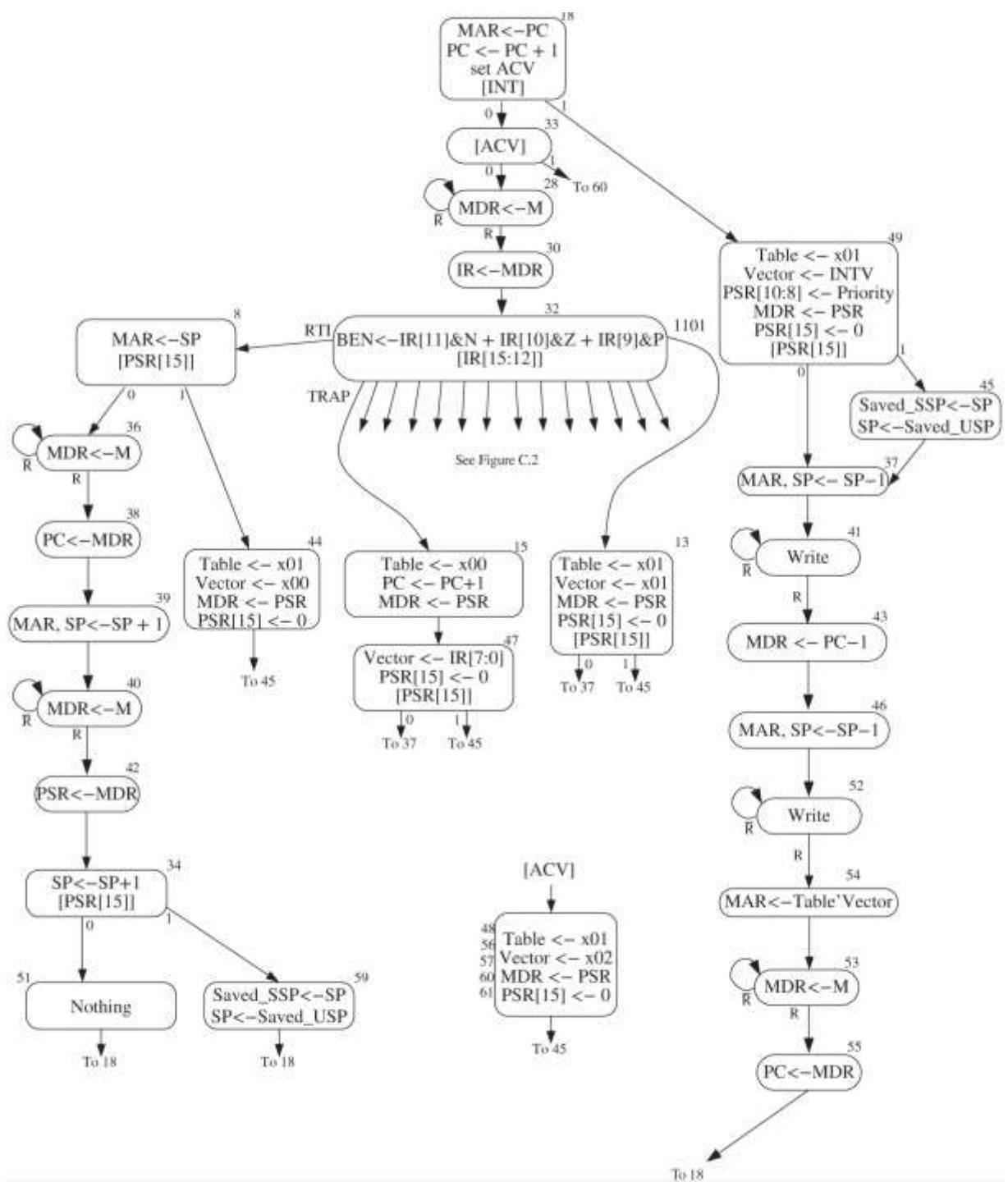
LC-3 as a Finite State Machine (see Patt Fig C.2 3rd ed.)



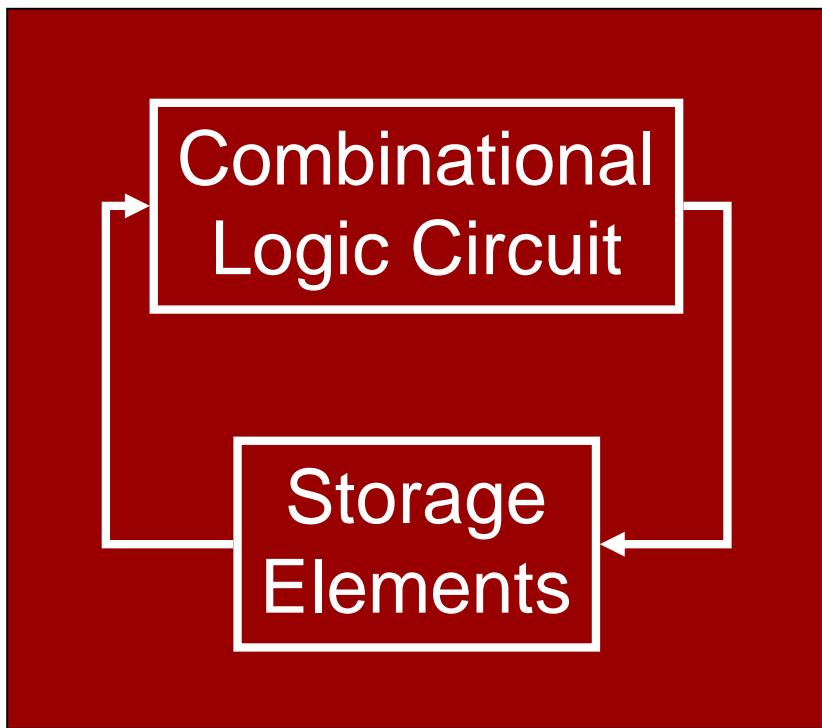
The Rest of the FSM; Part Fig C.7 Exception, Interrupts, Traps

This is the rest
of the FSM
diagram.

The only thing
you might need
are the TRAP
and RTI states



Sequential Logic Circuit (FSM)



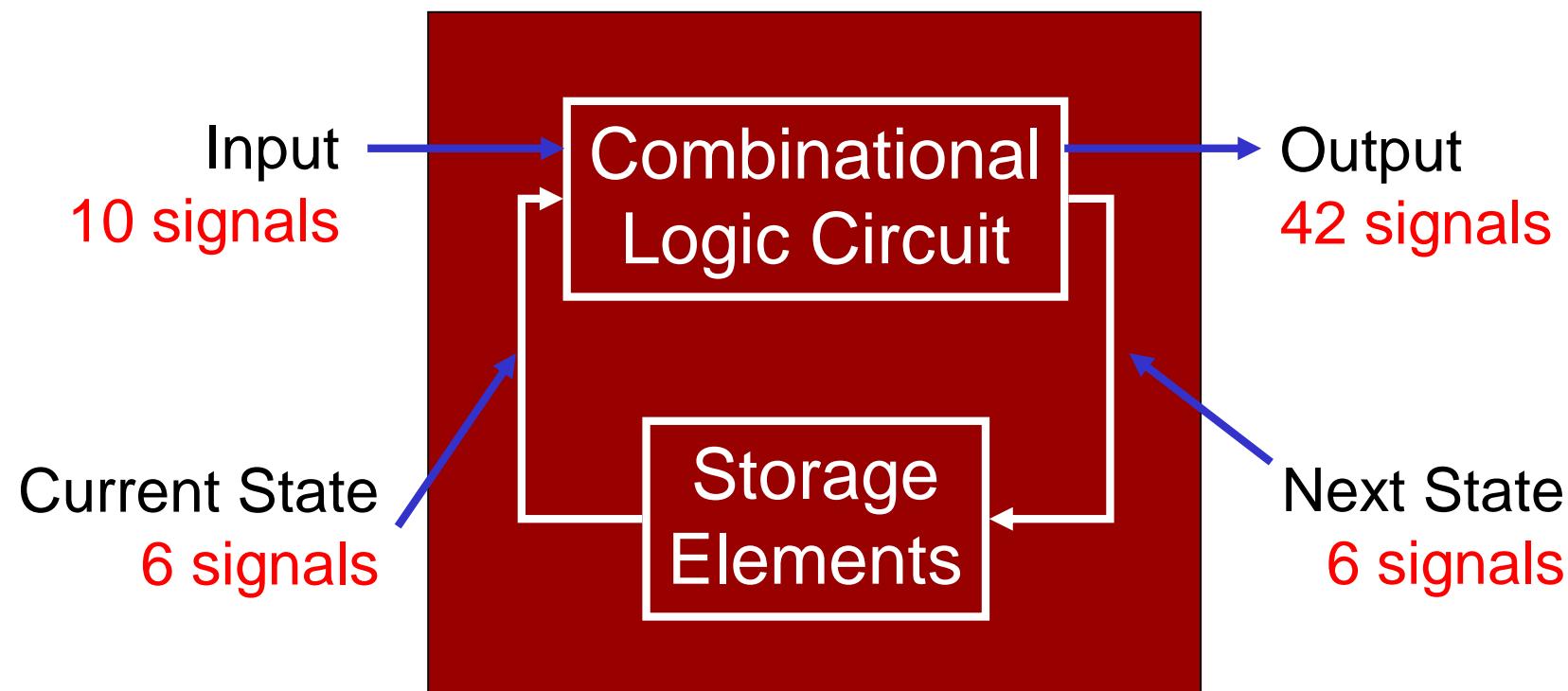
Question

We are given the diagram for a reasonably complex Finite State Machine. What is the best thing to do next in order to construct it?

- A. Build a K-map
- B. Build a truth table
- C. Build a set of Boolean equations
- D. Assemble a circuit



LC-3 FSM (Controller)



- ↗ How shall we implement this monster?
- ↗ Shall we start with a truth table?
- ↗ We have a state diagram with ≤ 64 states (hence the 6-bit Next State)
- ↗ How big will this truth table be?
- ↗ $10+6$ Inputs means 2^{16} rows

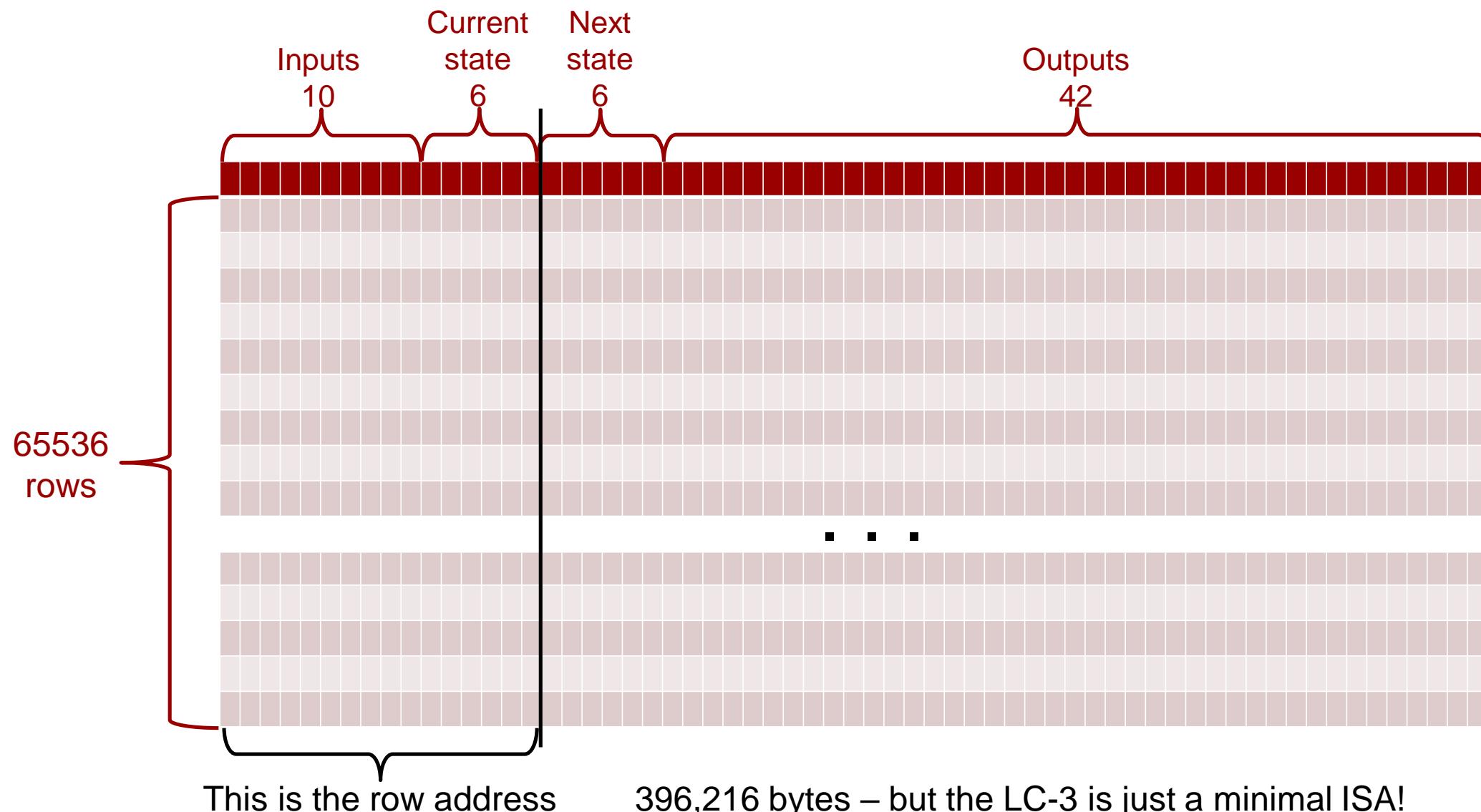
This Would Be Our Truth Table

Inputs	Current state	Next state	Outputs
10	6	6	42
...
...
65536 rows			

What Next?

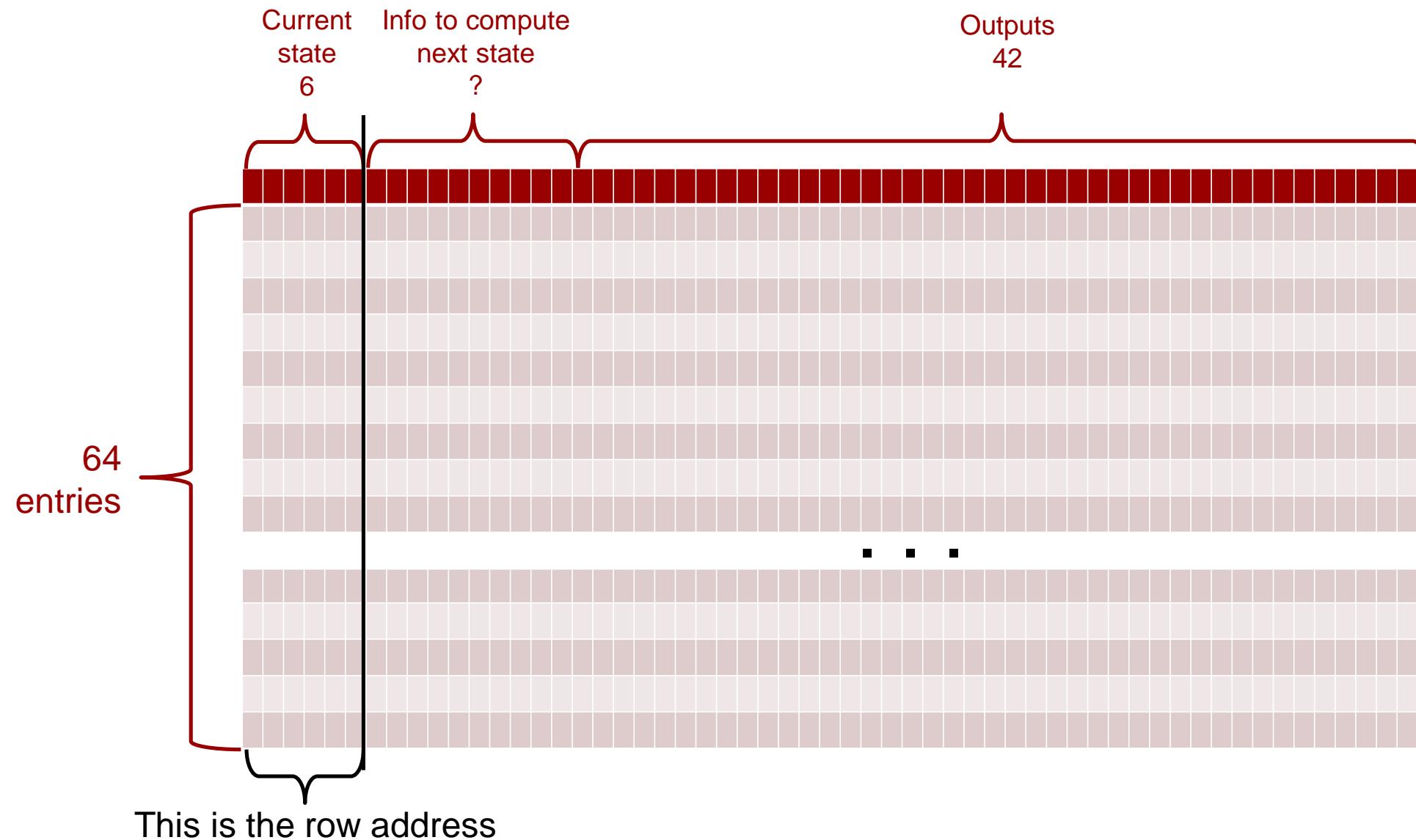
- ↗ Who volunteers to do the 48 8-dimensional 16-variable Karnaugh maps?
- ↗ Does this make you think our standard procedure for turning a truth table into an FSM is going to scale well?
- ↗ Do you think we could store our truth table in a read-only memory (ROM)?
- ↗ If we sort the rows in numerical order by the 16 Input bits, the Input bits would be the same as the memory address for that row
- ↗ Then we could look up our Output values and Next State with a single memory access! And we could just hook up our Output lines and Next State to the output from the memory!
- ↗ Wouldn't have to calculate and implement all that combinational logic!
- ↗ By the way, we wouldn't have to store the first 16 bits since it matches the address and we only use the Inputs to choose the row with the appropriate Outputs

Using a ROM for Our Truth Table



- ↗ That seems like a lot of memory for a small ISA; how is that going to scale?
- ↗ Another scaling problem, right?
- ↗ Recall that in the truth table, the Outputs (not including the Next State) are the same for all rows with the same Current State (because our Outputs are associated with a state, not the transition)
- ↗ What if we could compute the Next State in a different way? Then we'd only need one row of outputs for each FSM state! 64 rows instead of 65536!

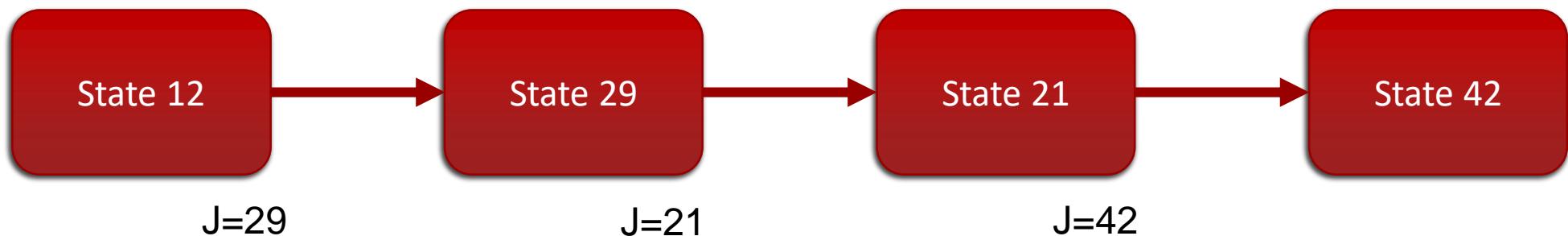
Now We Need to Compute Next State From the Inputs



A Work-Around

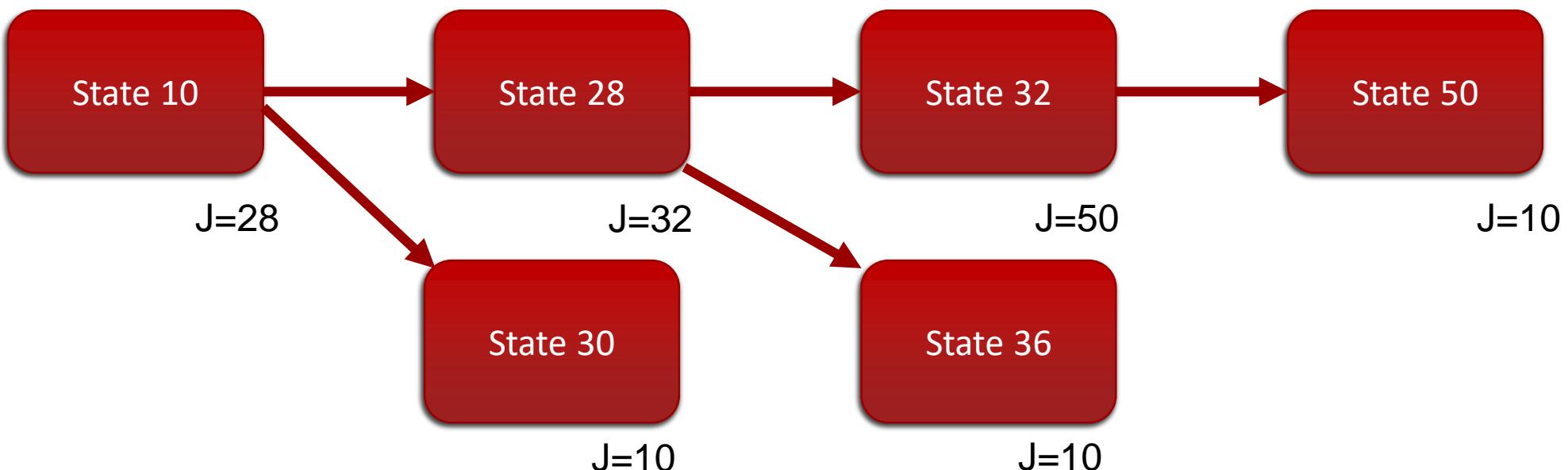
- Let's use 6 bits in each row for Next State (because most of the time there's only one state that can follow our state)?
 - We'll call this value $J[0:5]$

Here's an example; doesn't the state diagram look mostly like this?

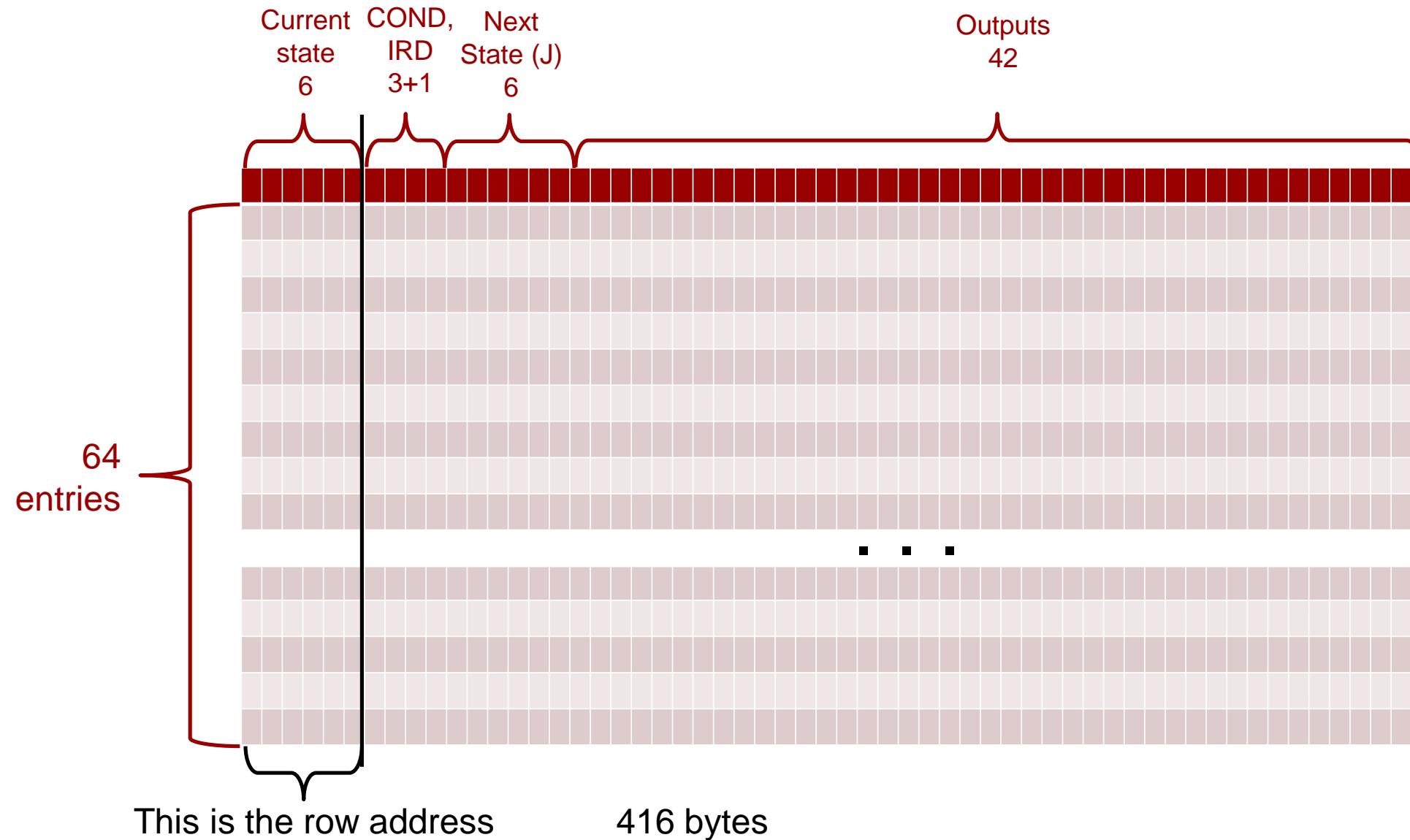


What If More than One State Can Follow Our State?

- First we'll agree to allow only 2 states to follow a state, at least most of the time (we can always add another state to choose between two more states)
- Then we add 4 more bits to each row into which we code the decisions when a state can be followed by one of two states
 - We'll call those values IRD[0] and COND[0:2]
 - If they're both zero, we'll just use the value of J for Next State



What Would our Control Store Look Like?



The Four “Magic” Bits for Choosing Next State

- ↗ We are going to use the COND and IRD bits to help us choose a different next state from the one listed by J
- ↗ If the IRD bit is true, we are going to use the opcode in IR[15:12] as the Next State, ignoring J completely
- ↗ COND allows us to choose one of six conditions to be tested

000	Unconditional		100	Privilege mode	J[3]
001	Memory Ready	J[1]	101	Interrupt present	J[4]
010	Branch Enabled	J[2]	110	Access Control Violation	J[5]
011	Addressing Mode	J[0]	111	Not Used	

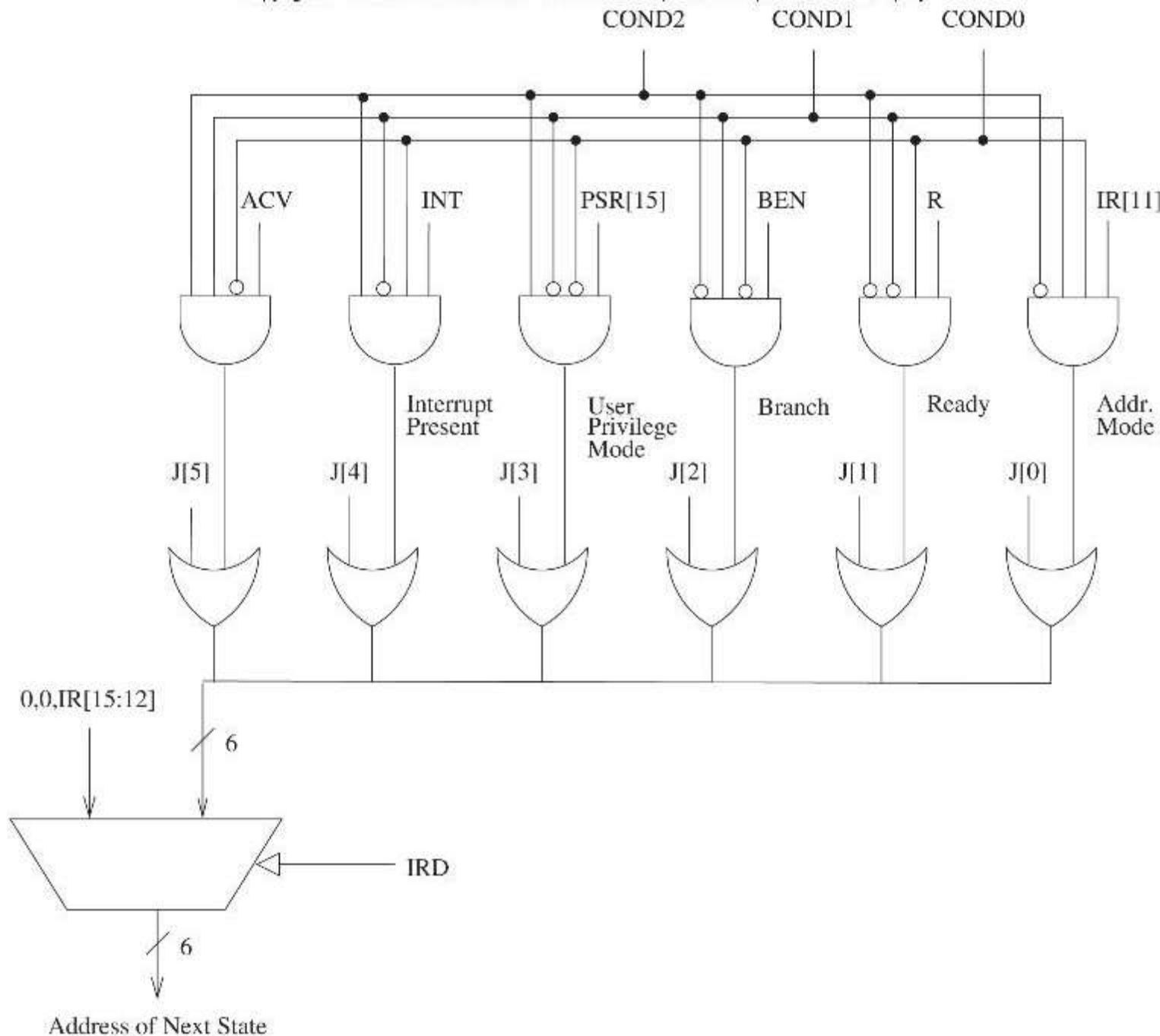
- ↗ When the condition is true, the indicated bit in J is set to 1
- ↗ That means in a state with COND set to something other than 000, there are exactly two possible next states:
 - ↗ J
 - ↗ J with one specific bit set to 1

Microsequencer

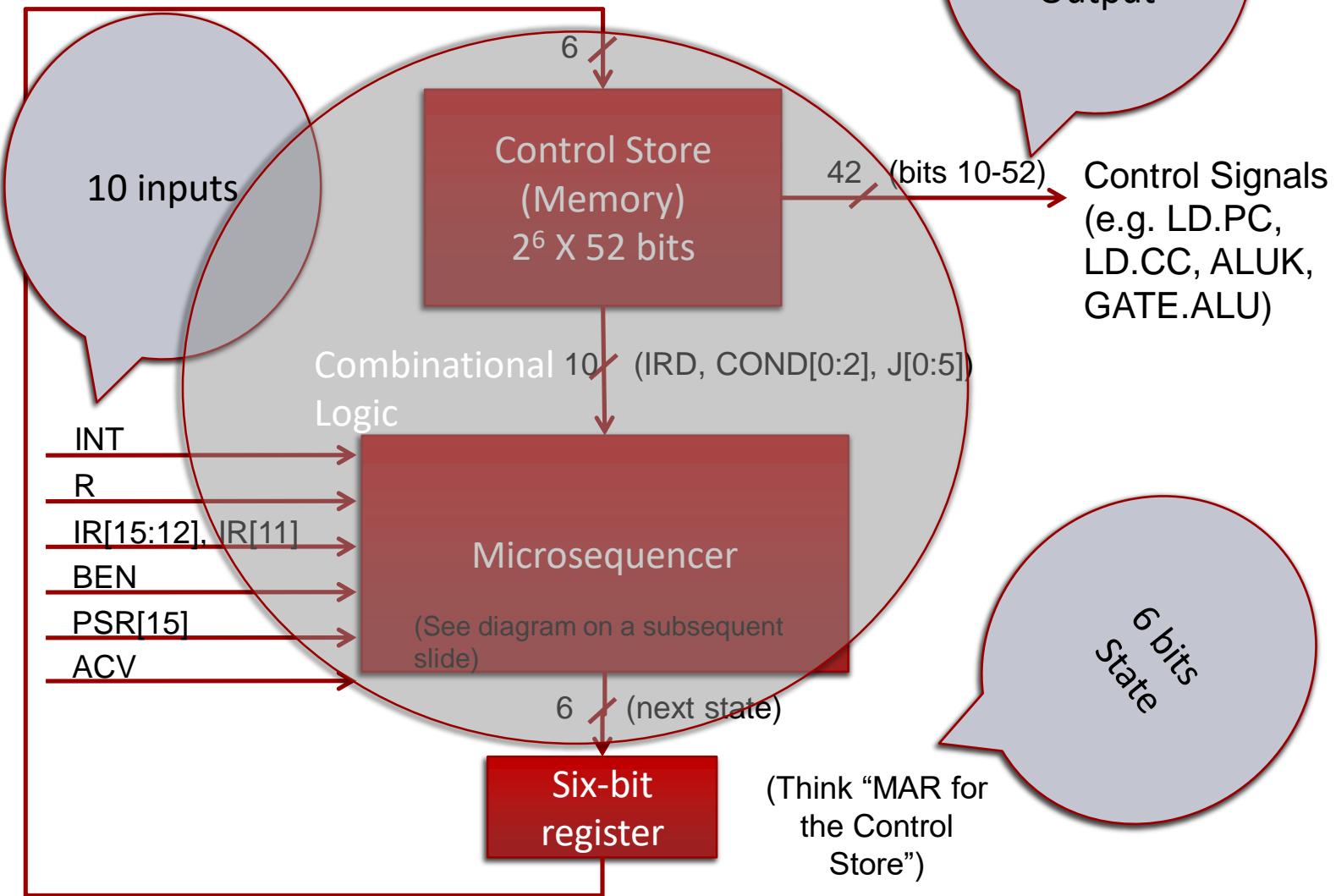
- ↗ We use the term **microsequencer** for the combinational circuit that chooses the next state from the 10 input lines and the current state
- ↗ For example, if the J=28 (011100) and COND=001, then if the memory is ready, the next state will be 30; otherwise it will be 28.
- ↗ If the J=18 (010010) and COND=010, then if the BEN register is true meaning the current machine instruction should branch, the next state will be 22; otherwise it will be 18
- ↗ See how simple that is. Each condition, when tested and true, sets a single bit in J to 1.
- ↗ Also note that the FSM is running a program described by the state diagram, but there is no Program Counter; each state must always specify the Next State – there is no implicit “next” state

The Microsequencer

Patt Fig C.6



The Control Structure



Question

In the LC-3 microsequencer, for a given Current State, how many possibilities are there for the Next State?

- A. 1
- B. 1 or 2
- C. 1, 2, or 3
- D. 1, 2, or 16



Data Path Control Signals

Patt Table C.1

Signal Name	Signal Values	
LD.MAR/1:	NO, LOAD	
LD.MDR/1:	NO, LOAD	
LD.IR/1:	NO, LOAD	
LD.BEN/1:	NO, LOAD	
LD.REG/1:	NO, LOAD	
LD.CC/1:	NO, LOAD	
LD.PC/1:	NO, LOAD	
LD.Priv/1:	NO, LOAD	
LD.Priority/1:	NO, LOAD	
LD.SavedSSP/1:	NO, LOAD	
LD.SavedUSP/1:	NO, LOAD	
LD.ACV/1:	NO, LOAD	
LD.Vector/1:	NO, LOAD	
GatePC/1:	NO, YES	
GateMDR/1:	NO, YES	
GateALU/1:	NO, YES	
GateMARMUX/1:	NO, YES	
GateVector/1:	NO, YES	
GatePC-1/1:	NO, YES	
GatePSR/1:	NO, YES	
GateSP/1:	NO, YES	
PCMUX/2:	PC+1 BUS ADDER	;select pc+1 ;select value from bus ;select output of address adder
DRMUX/2:	11.9 R7 SP	;destination IR[11:9] ;destination R7 ;destination R6
SR1MUX/2:	11.9 8.6 SP	;source IR[11:9] ;source IR[8:6] ;source R6

Data Path Control Signals (cont)

Patt Table C.1

ADDR1MUX/1:	PC, BaseR	
ADDR2MUX/2:	ZERO offset6 PCoffset9 PCoffset11	;select the value zero ;select SEXT[IR[5:0]] ;select SEXT[IR[8:0]] ;select SEXT[IR[10:0]]
SPMUX/2:	SP+1 SP-1 Saved SSP Saved USP	;select stack pointer+1 ;select stack pointer-1 ;select saved Supervisor Stack Pointer ;select saved User Stack Pointer
MARMUX/1:	7.0 ADDER	;select ZEXT[IR[7:0]] ;select output of address adder
TableMUX/1:	x00, x01	
VectorMUX/2:	INTV Priv.exception Opc.exception ACV.exception	
PSRMUX/1:	individual settings, BUS	
ALUK/2:	ADD, AND, NOT, PASSA	
MIO.EN/1:	NO, YES	
R.W/1:	RD, WR	
Set.Priv/1:	0 1	;Supervisor mode ;User mode

Contents of the Control Store

<i>J</i>	<i>IRD</i>	<i>Cond</i>	<i>LD.MAR</i>	<i>LD.MDR</i>	<i>LD.IR</i>	<i>LD.BEN</i>	<i>LD.REC</i>	<i>LD.CC</i>	<i>LD.PC</i>	<i>LD.Priv</i>	<i>LD.SavedSSP</i>	<i>LD.SavedUSP</i>	<i>LD.Vector</i>	<i>LD.Priority</i>	<i>LD.ACV</i>	<i>GatePC</i>	<i>GateMDR</i>	<i>GateALU</i>	<i>GateMARMUX</i>	<i>GateVector</i>	<i>GatePC-1</i>	<i>GatePSR</i>	<i>GateSP</i>	<i>PCMUX</i>	<i>DRMUX</i>	<i>SRIMUX</i>	<i>ADDRIMUX</i>	<i>ADDR2MUX</i>	<i>SPMUX</i>	<i>MARMUX</i>	<i>TableMUX</i>	<i>VectorMUX</i>	<i>PSRMUX</i>	<i>ALUK</i>	<i>MIO.EV</i>	<i>R.W</i>	<i>Sel.Priv</i>
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000000 (State 0)					
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000001 (State 1)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000010 (State 2)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000011 (State 3)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000100 (State 4)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000101 (State 5)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000110 (State 6)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	000111 (State 7)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001000 (State 8)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001001 (State 9)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001010 (State 10)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001011 (State 11)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001100 (State 12)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001101 (State 13)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001110 (State 14)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	001111 (State 15)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	010000 (State 16)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	010001 (State 17)						
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	010010 (State 18)						

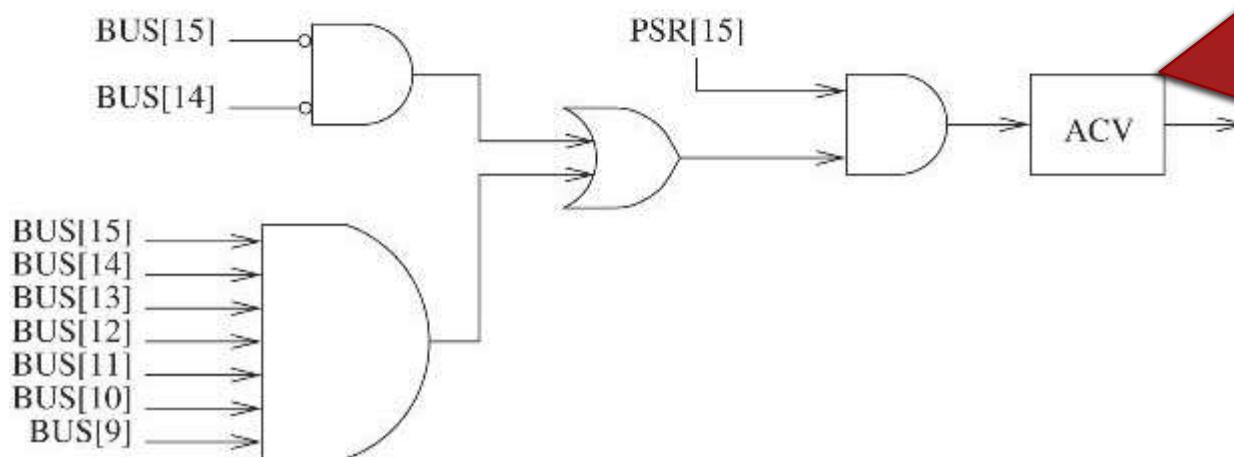
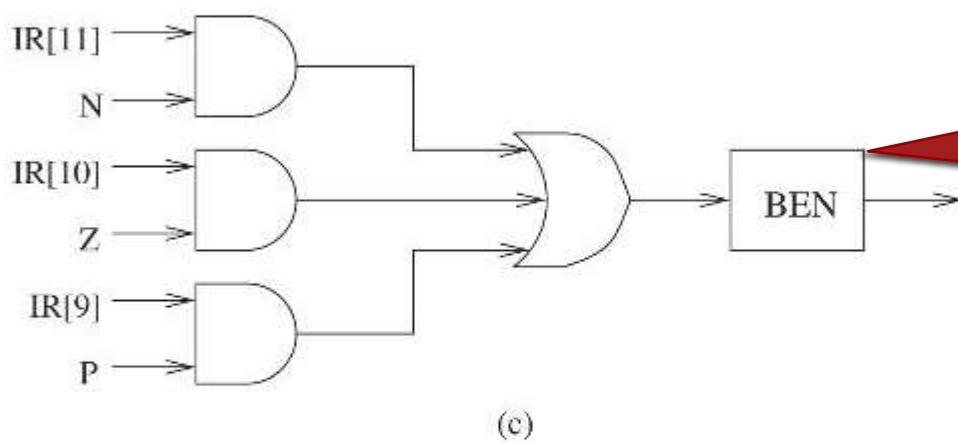
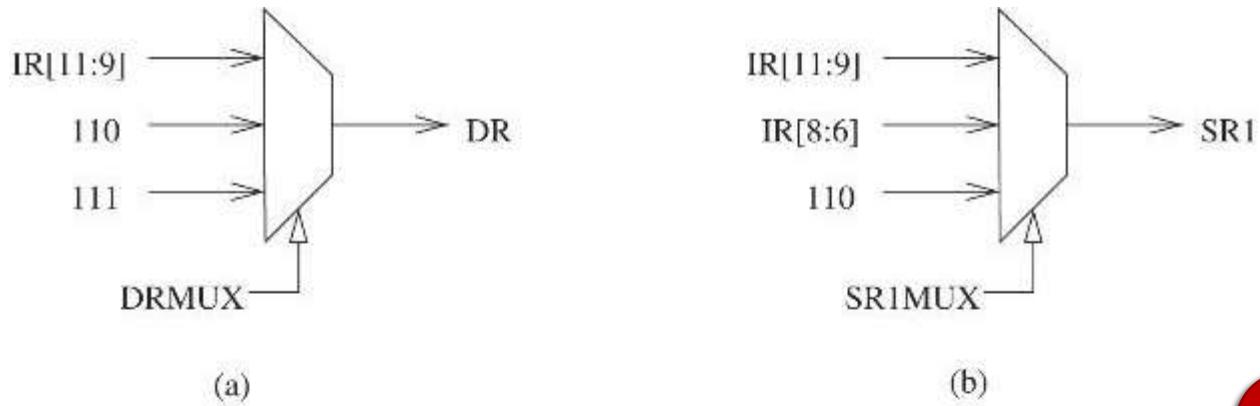
First 16 States of LC-3 Microcode

State	Next State	Control Signals (non-zero)
000000 (State 0)	COND=2 J=18	
000001 (State 1)	J=18	LD.REG LD.CC GateALU SR1MUX=1
000010 (State 2)	J=35	LD.MAR LD.ACV GateMARMUX ADDR2MUX=1 ADDR2MUX=2 MARMUX=1
000011 (State 3)	J=23	LD.MAR LD.ACV GateMARMUX ADDR2MUX=1 ADDR2MUX=2 MARMUX=1
000100 (State 4)	COND=3 J=20	
000101 (State 5)	J=18	LD.REG LD.CC GateALU SR1MUX=1 ALUK=1
000110 (State 6)	J=35	LD.MAR LD.ACV GateMARMUX ADDR1MUX=1 ADDR2MUX=2 ADDR2MUX=1 MARMUX=1
000111 (State 7)	J=23	LD.MAR LD.ACV GateMARMUX ADDR1MUX=1 ADDR2MUX=2 ADDR2MUX=1 MARMUX=1
001000 (State 8)	COND=4 J=36	LD.MAR GateALU SR1MUX=2 ALUK=3
001001 (State 9)	J=18	LD.REG LD.CC GateALU SR1MUX=1 ALUK=2
001010 (State 10)	J=17	LD.MAR LD.ACV GateMARMUX ADDR2MUX=1 ADDR2MUX=2 MARMUX=1
001011 (State 11)	J=19	LD.MAR LD.ACV GateMARMUX ADDR2MUX=1 ADDR2MUX=2 MARMUX=1
001100 (State 12)	J=18	LD.PC GateALU PCMUX=1 SR1MUX=1 ALUK=3
001101 (State 13)	COND=4 J=37	LD.MDR LD.Priv LD.Vector GatePSR TableMUX=1 VectorMUX=2
001110 (State 14)	J=18	LD.REG GateMARMUX ADDR2MUX=1 ADDR2MUX=2 MARMUX=1
001111 (State 15)	J=47	LD.MDR LD.PC LD.Vector GatePSR

You can find this information in figureC9v3.xls

Omissions From the Data Path

Diagram



Branch ENabled

(BEN)

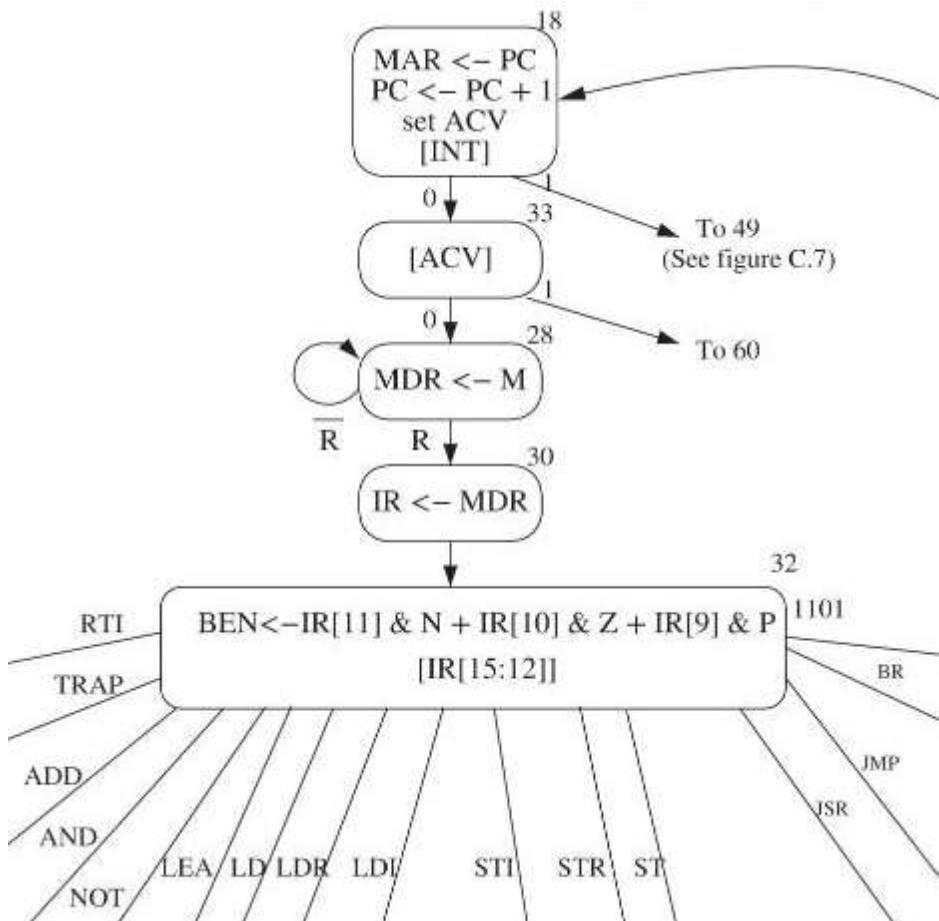
Register storing the answer to "Should we branch"

ACcess Violation

(ACV)

Register storing the answer to "Is the CPU in user state (PSR[15]) and (address on the bus < 0x3000 or >=0xFE00)"

Fetch Stage Example



- ↗ State 18 starts the fetch
 - ↗ MAR \leftarrow PC
 - ↗ PC \leftarrow PC + 1
 - ↗ Test PC value for Access Violation (out of bounds)
 - ↗ COND5 and J=33 (if interrupt go to 33+16=49 else 33)
- ↗ State 33 checks for an access violation
 - ↗ COND6 and J=28 (if ACV go to 28+32=60 else 28)
- ↗ State 28 waits for read complete
 - ↗ Assert MEM.EN
 - ↗ MDR \leftarrow MEM[MAR]
 - ↗ COND1 and J=28 (if R go to state 28+2=30 else 28)
- ↗ State 30 moves the instruction to IR
 - ↗ IR \leftarrow MDR
 - ↗ COND0 and J=32 so next state is 32

Fetch Stage Microcode

State	IRD	CONDx					
(State 18)	0	5	J=33	LD.MAR	LD.PC	GatePC	LD.ACV
(State 33)	0	6	J=28				
(State 28)	0	1	J=30	LD.MDR	MEM.EN		
(State 30)	0	0	J=32	LD.IR	GateMDR		

The Next State is a possibly modified version of J

COND6 says “Set bit 5 (+32) in J if ACV is true

COND5 says “Set bit 4 (+16) in J if an interrupt is waiting”

COND1 says “Set bit 1 (+2) in J if memory is Ready

COND0 says “Use unmodified J”

You can find this information in figureC9v3.xls

Branch Example

- ↗ State 32 does the decode
 - ↗ $BEN \leftarrow IR[11] \& N \mid IR[10] \& Z \mid IR[9] \& P$
 - ↗ State 32 has $IRD=1$ for next state
- ↗ State 0 is chosen for a BRx instruction
 - ↗ State 0 has $J=18$ and COND2 for next state
 - ↗ The next state will be 18 if BEN is 0 or 22 ($18 + 4$) if BEN is 1
- ↗ State 22 changes the PC
 - ↗ $PC \leftarrow PC + off9$
 - ↗ State has $J=18$ and COND0
 - ↗ So next state is 18
- ↗ State 18 starts the fetch cycle again

Decode/Execute Microcode for BR

State	IRD	CONDx						
(State 32)	1	0	J=0	LD.BEN				
(State 0)	0	2	J=18					
(State 22)	0	0	J=18	LD.PC	GateMARMUX	PCMUX=2	ADDR2MUX=2	

- ↗ IRD=1 says use IR[15:12] as the Next State
COND2 says “Set bit 2 (+4) in J if BEN is set”
- ↗ FYI, there is a single-bit BEN flip-flop with write-enable signal LD.BEN; this register is where the calculation of BEN is stored

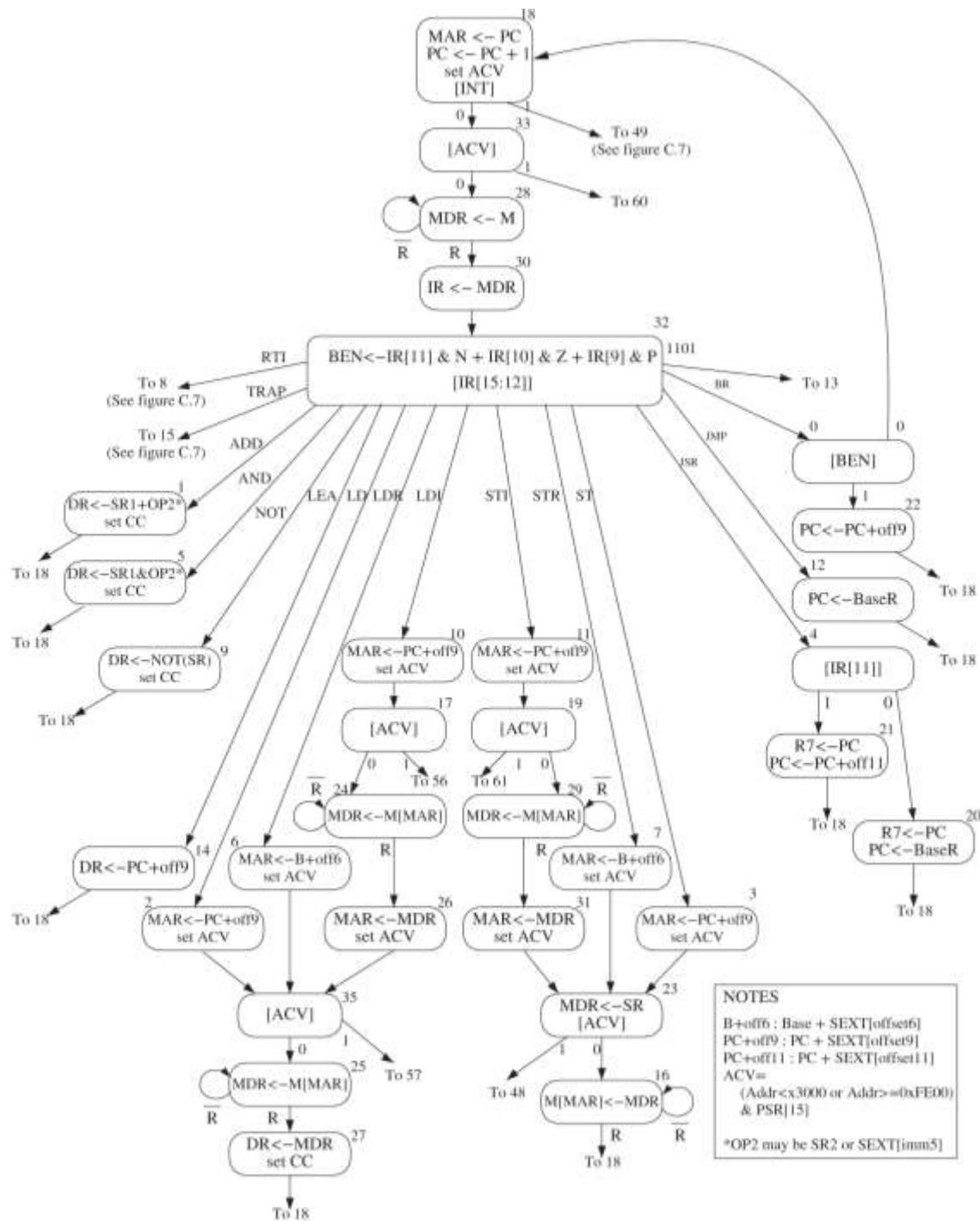
You can find this information in figurec9v2.xls

Microcode?

Note that the state diagram shows the register transfers done in each word/state in the microcode!

*In other words,
this is the
microcode in a
readable form!*

Fig C.3



Questions?

Which Do You Think Came First?

- ↗ (a) The data path diagram
- ↗ (b) The FSM graph
- ↗ (c) The microcode
- ↗ (d) The instruction set

- ↗ Why?

- The ISA: Overview
 - Memory, Registers, Instruction Set, Opcodes, Datatypes, Addressing Modes, Condition Codes
- Instruction Set
 - Datapath, Control signals to implement each instruction
 - ALU, Data movement (load/store), and Control Instructions
 - Addressing Modes
- The Microsequencer and Microcode

Assembly

Chapter 7



- Assembly Language
- Assembler Directives
- Labels (help for programmers with PC-rel)
- Aliases / Pseudo-Ops (HALT, RET, etc.)
- The Assembly Process
 - Symbol Table

Assembly Language

- ↗ Low level language
- ↗ Dependent on ISA
- ↗ Typically each instruction line in assembly language produces a single machine instruction
- ↗ Contrast with high level languages
 - ↗ FORTRAN, C, etc.
 - ↗ Typically converted to assembly!
- ↗ User friendly (compared to what, you ask?)
 - ↗ Mnemonics, not binary numbers
 - ↗ Names for memory addresses and opcodes
- ↗ Does a bit more than just the machine instructions

- Choices of approach
 - Assembly language programming for non-programmers
 - Assembly language programming for experienced high-level language programmers
- Choice of goal
 - Fluency in the language for a career in assembly language programming
 - A basic competency in assembly language for use as a tool in debugging, computer architecture, and optimization

- Choices of approach
 - Assembly language programming for non-programmers
 - Assembly language programming for experienced high-level language programmers
- Choice of goal
 - Fluency in the language for a career in assembly language programming
 - A basic competency in assembly language for use as a tool in debugging, computer architecture, and optimization

An Example

- ↗ Is there an LC-3 instruction to multiply a number already in memory by 6?
- ↗ Can we write code to do it?
- ↗ How?

Times 6

```
; Multiply a number by 6
.orig x3050
LD    R1, SIX          ; R1 is loop counter
LD    R2, NUMBER
AND   R3, R3, #0        ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN ADD   R3, R3, R2  ; Summing into R3
        ADD   R1, R1, #-1 ; Dec loop counter
        BRP  AGAIN
HALT

;
NUMBER .blkw 1
SIX   .fill x0006
.end
```

Assembler Directives

```
; Multiply a number by 6
.orig x3050
LD    R1, SIX           ; R1 is loop counter
LD    R2, NUMBER
AND   R3, R3, #0         ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN ADD   R3, R3, R2  ; Summing into R3
ADD   R1, R1, #-1       ; Dec loop counter
BRP   AGAIN
HALT

; 
NUMBER .blkw 1
SIX    .fill x0006
.end
```

Assembler directives (or pseudo-ops) tell the assembler to do something

Labels

```
; Multiply a number by 6
.orig x3050
LD R1, SIX ; R1 is loop counter
LD R2, NUMBER
AND R3, R3, #0 ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN    ADD R3, R3, R2 ; Summing into R3
        ADD R1, R1, #-1 ; Dec loop counter
        BRP AGAIN
HALT

;
NUMBER .blkw 1
SIX    .fill x0006
.end
```

Tags (or labels) lets you assign names to memory addresses, even if they move around!

Opcodes and Operands

```
; Multiply a number by 6
.orig x3050
LD    R1, SIX          ; R1 is loop counter
LD    R2, NUMBER
AND   R3, R3, #0        ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN ADD   R3, R3, R2      ; Summing into R3
      ADD   R1, R1, #-1     ; Dec loop counter
      BRP  AGAIN
HALT

;
NUMBER .blkw 1
SIX   .fill x0006
.end
```

Must have Opcode and Operands

Options

```
; Multiply a number by 6
.orig x3050
LD    R1, SIX          ; R1 is loop counter
LD    R2, NUMBER
AND   R3, R3, #0        ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN
    ADD  R3, R3, R2      ; Summing into R3
    ADD  R1, R1, #-1      ; Dec loop counter
    BRP AGAIN
HALT

;
NUMBER .blkw 1
SIX   .fill x0006
.end
```

Label (Tag) and Comments optional

Number Bases

```
; Multiply a number by 6
.orig x3050
LD R1, SIX ; R1 is loop counter
LD R2, NUMBER
AND R3, R3, #0 ; Clr R3, will hold product

; Do until R1 <= 0
AGAIN ADD R3, R3, R2 ; Summing into R3
ADD R1, R1, #-1 ; Dec loop counter
BRP AGAIN
HALT

;
NUMBER .blkw 1
SIX .fill x0006
.end
```

Decimal	#
Binary	b
Hex	x

Memory for Data

```
; Multiply a number by 6
.orig x3050
LD R1, SIX ; R1 is loop counter
LD R2, NUMBER
AND R3, R3, #0 ; Clr R3, will hold product
```

```
; Do until R1 <= 0
AGAIN ADD R3, R3, R2 ; Summing into R3
ADD R1, R1, #-1 ; Dec loop counter
BRP AGAIN
HALT
```

```
; NUMBER .blkw 1
SIX .fill x0006
.end
```

More than just code, need to worry about memory allocation

What Gets Assembled (e.g. What complx shows)

<u>Mem Addr</u>	<u>Hex</u>	<u>Dec</u>	
3050:	x2207	8711	LD R1, SIX
3051:	x2405	9221	LD R2, NUMBER
3052:	x56E0	22240	AND R3, R3, #0
3053:	x16C2	5826	AGAIN
3054:	x127F	4735	ADD R3, R3, R2
3055:	x03FD	1021	ADD R1, R1, #-1
3056:	xF025	-4059	BRP AGAIN
3057:	x0000	0	HALT
3058:	x0006	6	NUMBER

x5 = 5, so x3052 + 5 = x3057 or “NUMBER”

What Gets Assembled (e.g. What complx shows)

<u>Mem Addr</u>	<u>Hex</u>	<u>Dec</u>	
3050:	x2207	8711	LD R1, SIX
3051:	x2405	9221	LD R2, NUMBER
3052:	x56E0	22240	AND R3, R3, #0
3053:	x16C2	5826	AGAIN
3054:	x127F	4735	ADD R3, R3, R2
3055:	x03FD	1021	ADD R1, R1, #-1
3056:	xE025	-4059	BRP AGAIN
3057:	x0000	0	HALT
3058:	x0006	6	NUMBER
		—	NOP ■
		—	NOP ■

x1FD=-3, so x3056 – 3 = x3053 or “AGAIN”

Instructions

```
; Multiply a number by 6
.orig x3050
; R1 = ct, R2 = multiplier, R3 = prod

LD    R1, SIX      ; ct = 6
LD    R2, NUMBER   ; multiplier = NUMBER
AND   R3, R3, #0   ; prod = 0
; do {
AGAIN ADD   R3, R3, R2   ;     prod += multiplier
ADD   R1, R1, #-1   ;     ct -= 1
BRP  AGAIN        ; } while (ct > 0);
HALT

;
NUMBER .blkw 1
SIX    .fill x0006
.end
```

Any clearer?

Change the Algorithm?

```
; Multiply a number by 6
.orig x3050

LD    R2, NUMBER ; R2 = NUMBER
;
ADD  R2, R2, R2 ; R2 = R2*2
ADD  R3, R2, R2 ; R3 = R2*4
ADD  R3, R3, R2 ; R3 = R2*4 + R2*2
;
HALT
;
NUMBER .blkw 1
.end
```

Any better?

Question

An assembler directive

- A. Causes one machine instruction to be assembled
- B. Causes more than one machine instruction to be assembled
- C. Tells the assembler to do something other than assemble a machine instruction
- D. None of the above



Today's number is 22,521

Pseudo-ops (Assembler Directives)

- ↗ **.orig**
 - ↗ Where to put the data to be assembled
- ↗ **.fill**
 - ↗ Initialize one location
- ↗ **.blkw n**
 - ↗ Set aside n words in memory
- ↗ **.stringz "sample"**
 - ↗ Initialize 7 locations (sample + 0 word)
- ↗ **.end**
 - ↗ End of assembly program or block

- Reserve and fill a word with a numeric value, optionally naming it

- Reserve and optionally name the beginning of an area of memory

.blkw Motivation

Code

.orig x3000

A .fill x3012

B .fill 21

C .fill 0

Memory

x3000: x3012

x3001: x0015

x3002: x0000

Suppose we want some arrays?

```
ARR_A .fill 0
```

```
    .fill 0
```

```
ARR_B .fill 0
```

```
    .fill 0
```

```
    .fill 0
```

Suppose we want an array?

```
ARR_A.fill 0
```

```
    .fill 0
```

```
ARR_B.fill 0
```

```
    .fill 0
```

```
    .fill 0
```

Now suppose
we want ARR_A
to have 1000
Elements!

Suppose we want an array?

ARR_A .blkw 1000

ARR_B .blkw 3

**ARR_A has
1000
Elements**

1000 words of memory are
blocked out, or reserved,
for ARR_A

- Store an ASCII string in memory with a zero terminator, optionally naming it

Consider

message

.fill 'H'

.fill 'e'

.fill 'l'

.fill 'l'

.fill 'o'

.fill 0

Consider

```
message .stringz "Hello"
```

What does this assembler directive do?

C .blkw 20

- A. Sets C to the current address and causes a word containing decimal 20 to be placed in memory at the current address
- B. Sets C to the current address and causes a 20 words containing decimal 20 to be placed in memory starting at the current address
- C. Sets C to the current address and reserves 20 words of memory starting at C
- D. Sets C to the current address + 1 and reserves 20 words of memory starting at C

Today's number is
36,360



The Assembly Process

↗ Objective

- ↗ Translate the AL (Assembly Language) program into ML (Machine Language).
- ↗ Each AL instruction yields one ML instruction word.
- ↗ Interpret pseudo-ops correctly.

↗ Problem

- ↗ An instruction may reference a label.
- ↗ If the label hasn't been encountered yet, the assembler can't form the instruction word

↗ A Solution

- ↗ Two-pass assembly

Two-Pass Assembly - 1

- First Pass - generating the symbol table
 - Scan each line
 - Keep track of current address (location counter)
 - Increment by 1 for each instruction
 - Adjust the location counter as required for any pseudo-ops (e.g. .fill or .stringz, etc.)
 - For each label
 - Enter it into the symbol table along with the current address (location counter)
 - Stop when .end is encountered

Symbol Table Example

Symbol	Address
AGAIN	X3053
NUMBER	X3057
SIX	X3058

```
; ; Program to multiply a number by six
; ; .orig x3050
; ; LD R1, SIX
; ; LD R2, NUMBER
; ; AND R3, R3, #0
; ; ; The inner loop
; ; ; x3053 AGAIN ADD R3, R3, R2
; ; ; x3054 ADD R1, R1, #-1
; ; ; x3055 BRP AGAIN
; ; ; x3056 HALT
; ; ; x3057 NUMBER .blkw 1
; ; ; x3058 SIX .fill x0006
; ; ; .end
```

Two-Pass Assembly - 2

- Second Pass - generating the ML program
 - Scan each line again
 - Translate each AL instruction into ML
 - Look up symbols in the symbol table
 - Ensure that labels are no more than +256 / -255 lines from PCoffset9 instructions
 - Calculate operand field for the instruction
 - Update the current address (location counter)
 - Fill memory locations as directed by pseudo-ops
 - Stop when .end is encountered

Assembled code

➤ Using the earlier example:

Symbol	Address
AGAIN	X3053
NUMBER	X3057
SIX	X3058

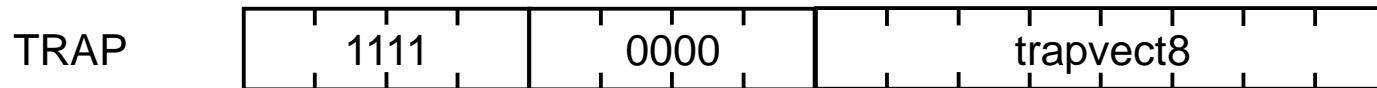
```
x3050 0010 001 0 0000 0111 ; LD R1, SIX
x3051 0010 010 0 0000 0101 ; LD R2, NUMBER
x3052 0101 011 011 1 00000 ; AND R3, R3, #0
x3053 0001 011 011 0 00 010 ; ADD R3, R3, R2
x3054 0001 001 001 1 11111 ; ADD R1, R1, #-1
x3055 0000 001 1 1111 1101 ; BRP AGAIN
x3056 1111 0000 0010 0101 ; HALT
x3057
; .blkw 1
x3058 0000 0000 0000 0110 ; .fill x0006
```

The assembly symbol table

- A. Is created during the first pass of assembly
- B. Is created during the second pass of assembly
- C. Holds only symbols that label machine instructions
- D. Is needed only in the first pass of assembly



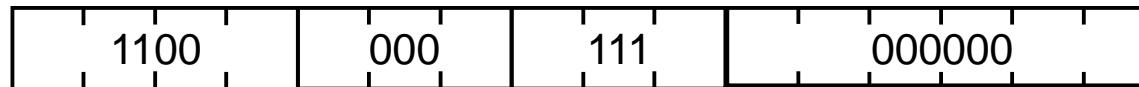
Aliases: TRAP pseudo-instruction



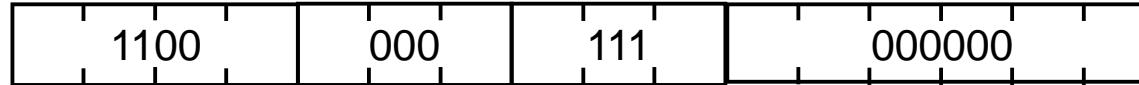
- The assembler will recognize aliases for certain predefined trap instructions:
 - TRAP x25 HALT ;Stop the CPU
 - TRAP x23 IN ;Input character from kybd
 - TRAP x21 OUT ;Print character on screen
 - TRAP x22 PUTS
 - TRAP x24 PUTSP
 - TRAP x20 GETC
- We'll talk a bit later about how traps are implemented!

Alias (Pseudo-Instructions)

RET



JMP R7



Same thing!

Thought Question

- ↗ Do we need all of these load instructions?

Need LD?

LD R1, DATA

...

DATA .fill 5

LEA R1, DATA

LDR R1, R1, #0

...

DATA .fill 5

Need LDI?

```
LDI R1, ADDR
```

```
...
```

```
ADDR .fill xFE00
```

```
LEA R1, ADDR
```

```
LDR R1, R1, #0
```

```
LDR R1, R1, #0
```

```
...
```

```
ADDR .fill xFE00
```

Questions?

- Assembly Language
- Assembler Directives
- Labels (help for programmers with PC-rel)
- Aliases / Pseudo-Ops (HALT, RET, etc.)
- The Assembly Process
 - Symbol Table

Assembly Programming



- ↗ Arithmetic Operations
 - ↗ How to subtract, OR, clear a register
- ↗ Control Instructions
 - ↗ Conditional Branches, Jump Instruction
 - ↗ Example
- ↗ Code Templates
 - ↗ IF, FOR, WHILE, DO WHILE
 - ↗ Examples

What can you do with ADD, AND and NOT?

- ↗ Which is all our ALU can do!

What can you do with ADD, AND and NOT

- ↗ Add
- ↗ And
- ↗ Not
- ↗ Subtract
- ↗ Or
- ↗ Clear a register
- ↗ Copy from one register to another
- ↗ Increment a register by n, $-16 \leq n \leq 15$

This is just enough for the LC-3 to be Turing-complete!

Subtract

```
;SUB R1, R2, R3 ;instruction does not exist  
;  
; how to subtract  
  
NOT R3, R3 ;flip the bits of R3  
  
ADD R3, R3, #1 ;add 1 to R3 ;now R3 is -R3  
  
ADD R1, R2, R3
```

OR

;OR R1, R2, R3 ;does not exist: R1=R2 | R3

; how to do OR ;use DeMorgan's Law

NOT R2, R2 ;R2 = ~R2

NOT R3, R3 ;R3 = ~R3

AND R1, R2, R3 ;R1 = ~R2 & ~R3

NOT R1, R1 ;R1 = ~ (~R2 & ~R3)

Other Operations

;CLR R1 ;e.g. set R1 = 0

AND R1, R1, #0 ;R1 = R1 & 0

;anything and zero is zero

;MOV R1, R2 ;e.g. R1 = R2

ADD R1, R2, #0 ;R1 = R2+0; R1 = R2

Control Instructions

BR	0000	n	z	p	PCoffset9
JMP	1100	000	BaseR	000000	
JSR	0100	1		PCoffset11	
JSRR	0100	0	00	BaseR	000000
RET	1100	000	111	000000	
RTI	1000		000000000000		
TRAP	1111	0000		trapvect8	

Conditional Branch

BR



What can we do with a BR?

What can we do with a BR?

- ↗ If then else
- ↗ For
- ↗ While
- ↗ Do while
- ↗ Conditional Branch
- ↗ Unconditional branch (BRNZP or BR)
- ↗ Never branch (NOP)

JMP Instruction

JMP



What can we do with a JMP?

What can we do with a JMP?

- ↗ Go To!!!!
- ↗ Branch long distances!!!

- ↗ We will rarely use JMP in this class
- ↗ BR is much more commonly used

Differences Between BR & JMP

↗ BRx

- ↗ Can branch on N, Z, and P conditions
- ↗ Can always branch (BR or BRnzp)
- ↗ Can never branch (NOP)
- ↗ Destination address is always PCoffset9
- ↗ Can't branch more than -256 to 255 words

↗ JMP

- ↗ Always branches
- ↗ Destination address **always** in a register
- ↗ Can branch to any memory address

How do we do IF using a BR (branch)?

- First do an operation to set the condition codes
- Then BR with the appropriate combination of NZP conditions in the instruction

Every Condition: Comparison with Zero

- ↗ A < B
- ↗ A == B
- ↗ A-B < B-B
- ↗ A-B == B-B
- ↗ A-B < 0
- ↗ A-B == 0
- ↗ N condition code
- ↗ Z condition code

Condition Codes

N Z P

Think: < == >

BR (branch conditions)

- < N
- <= NZ
- == Z
- != NP
- >= ZP
- > P
- Always NZP (we abbreviate BRnzp as just BR, branch always)
- Never (no condition codes), we use NOP (no operation)

Stylized Assembly Coding

- Assembly programming gives the programmer a LOT of freedom
 - Freedom to optimize performance whether needed or not
 - Freedom to write impossibly complex code
 - Freedom to make a zillion different errors
 - Freedom to debug everything in binary
- Is this always a good idea?
 - If you're just learning
 - If you don't want to make a career of it
 - If you recognize that a compiler/optimizer can do a better job
- What if we taught you some conventions that allow you to write and understand code much more quickly? Would you use them?

Our Suggested Convention: Act Like a Compiler

- ↗ Write your algorithm in a high-level language
- ↗ Write down where you are going to store your variables (register, static memory, stack offset, etc.)
 - ↗ Use comments for registers and stack offsets
 - ↗ Use assembler directives to reserve memory
- ↗ Copy your algorithm with ";" at the beginning of each line to create assembly language comments

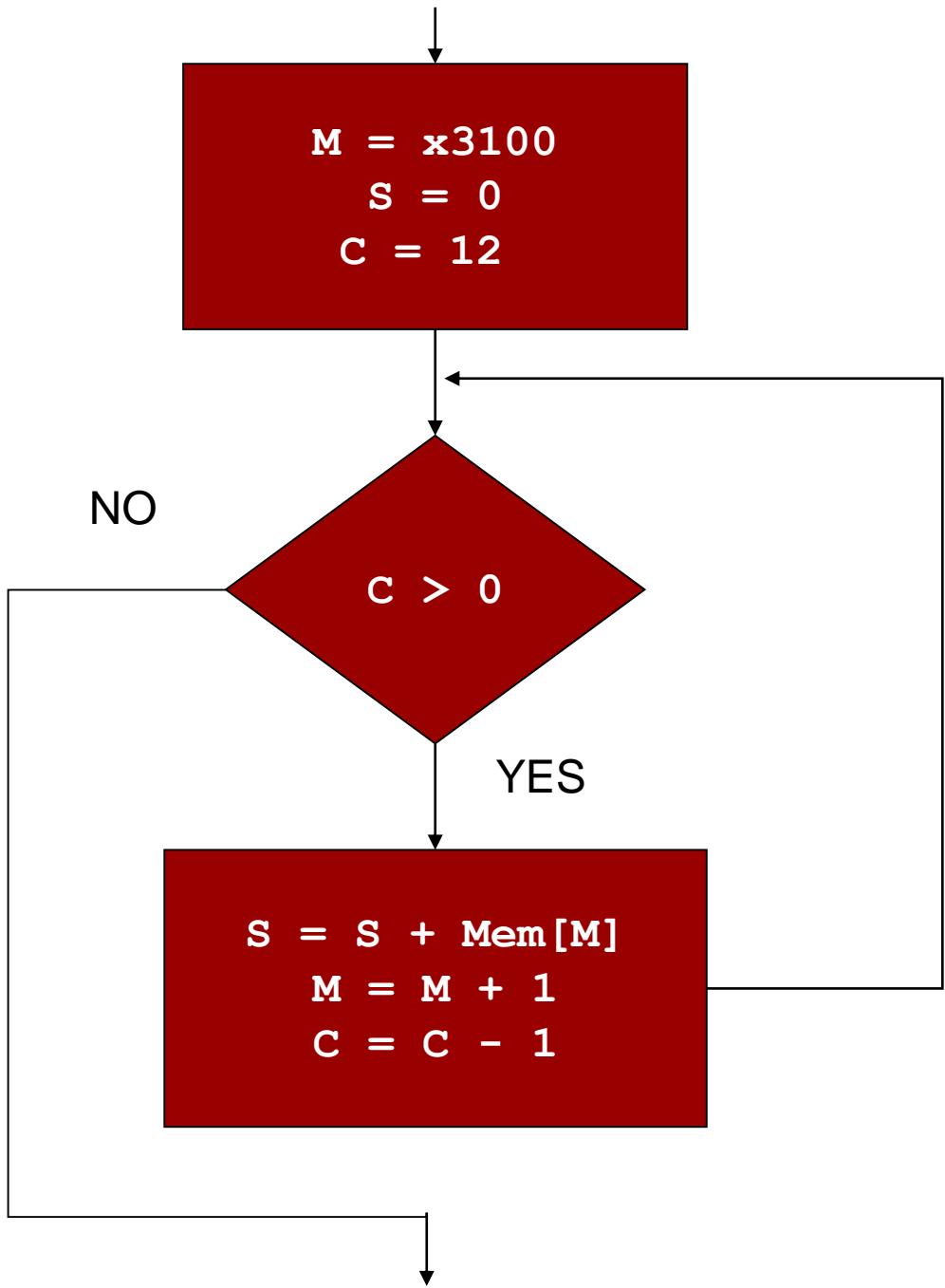
Our Suggested Convention: Act Like a Compiler

- Translate each statement into a stanza of the appropriate machine language instructions
 - For complex statements, like IF, WHILE, CALL, use templates!
 - Make sure that each stanza you translate is independent of your other stanzas
 - At the start, don't depend on the contents of temporary registers
 - At the start, don't depend on the condition codes
 - At the end, store your modified values where they belong
- This will NOT result in the most efficient code, but it will be correct code
 - plus an optimizer could easily make it efficient by throwing out the unneeded instructions you write, so who cares?

Example Program

- Write a program to be located at address x3000 which will sum up 12 integers located in memory addresses x3100-x310B

Sum a List of 12 Integers



Write Some Pseudo-Code (or Java or C or ...)

```
short Mem[65536];
int S = 0;
int M = x3100;
int C = 12;
while (C > 0) {
    S = S + Mem[M];
    C = C - 1;
    M = M + 1;
}
```

Turn Your Pseudo-Code Into Assembly Language Comments

```
; int S = 0;  
; int M = x3100;  
; int C = 12;  
; while (C > 0) {  
    ; S = S + Mem[M];  
    ; C = C - 1;  
    ; M = M + 1;  
}
```

Decide Where Variables Will Be Stored

; S is R1, M is R2, C is R3

; int S = 0;

; int M = x3100;

; int C = 12;

; while (C > 0) {

 ; S = S + Mem[M];

 ; C = C - 1;

 ; M = M + 1;

}; }

int S = 0

; S is R1, M is R2, C is R3

; int S = 0;

AND R1,R1,#0

; int M = x3100;

; int C = 12;

; while (C > 0) {

; S = S + Mem[M];

; C = C - 1;

; M = M + 1;

; }

int M = x3100

```
; S is R1, M is R2, C is R3  
; int S = 0;  
    AND    R1,R1,#0  
; int M = x3100;  
    LD     R2,MINIT  
; int C = 12;  
; while (C > 0) {  
    ; S = S + Mem[M];  
    ; C = C - 1;  
    ; M = M + 1;  
}  
;
```

MINIT .fill x3100

int C = 12

```
; S is R1, M is R2, C is R3
; int S = 0;
    AND    R1,R1,#0
; int M = x3100;
    LD     R2,MINIT
; int C = 12;
    AND    R3,R3,#0
    ADD    R3,R3,#12
; while (C > 0) {
    ; S = S + Mem[M];
    ; C = C - 1;
```

```
; M = M + 1;
; }
MINIT .fill    x3100
```

while (C > 0) {

; S is R1, M is R2, C is R3

; int S = 0;

AND R1,R1,#0

; int M = x3100;

LD R2,MINIT

; int C = 12;

AND R3,R3,#0

ADD R3,R3,#12

; while (C > 0) {

W1 ADD R3,R3,#0

BRnz ENDW1

; S = S + Mem[M];

; C = C - 1;

; M = M + 1;

; }

MINIT .fill x3100

} (end of the while)

```
; S is R1, M is R2, C is R3  
; int S = 0;  
    AND    R1,R1,0  
; int M = x3100;  
    LD     R2,MINIT  
; int C = 12;  
    AND    R3,R3,#0  
    ADD    R3,R3,12  
; while (C > 0) {  
W1    ADD    R3,R3,#0  
    BRnz   ENDW1
```

```
; S = S + Mem[M];  
; C = C - 1;  
; M = M + 1;  
; }  
BR W1  
ENDW1  
MINIT .fill x3100
```

$$S = S + \text{Mem}[M]$$

```
; S is R1, M is R2, C is R3  
; int S = 0;  
    AND    R1,R1,0  
; int M = x3100;  
    LD     R2,MINIT  
; int C = 12;  
    AND    R3,R3,#0  
    ADD    R3,R3,12  
; while (C > 0) {  
W1    ADD    R3,R3,#0  
    BRnz   ENDW1
```

```
; S = S + Mem[M];  
    LDR    R4,R2,#0  
    ADD    R1,R1,R4  
; C = C - 1;  
; M = M + 1;  
; }  
    BR    W1  
ENDW1  
MINIT .fill x3100
```

C = C - 1

```
; S is R1, M is R2, C is R3
; int S = 0;
    AND    R1,R1,#0
; int M = x3100;
    LD     R2,MINIT
; int C = 12;
    AND    R3,R3,#0
    ADD    R3,R3,#12
; while (C > 0) {
W1    ADD    R3,R3,#0
    BRnz   ENDW1
; S = S + Mem[M];
```

```
LDR    R4,R2,#0
ADD    R1,R1,R4
; C = C - 1;
    ADD    R3,R3,#-1
; M = M + 1;
; }
BR     W1
ENDW1
MINIT .fill  x3100
```

M = M + 1

```
; S is R1, M is R2, C is R3
; int S = 0;
    AND    R1,R1,#0
; int M = x3100;
    LD     R2,MINIT
; int C = 12;
    AND    R3,R3,#0
    ADD    R3,R3,#12
; while (C > 0) {
W1    ADD    R3,R3,#0
    BRnz   ENDW1
; S = S + Mem[M];
```

```
LDR    R4,R2,#0
ADD    R1,R1,R4
; C = C - 1;
ADD    R3,R3,#-1
; M = M + 1;
ADD    R2,R2,#1
; }
BR     W1
ENDW1
MINIT .fill  x3100
```

Add a Little Window Dressing

```
.orig x3000
; S is R1, M is R2, C is R3
; int S = 0;
    AND R1,R1,#0
; int M = x3100;
    LD   R2,MINIT
; int C = 12;
    AND R3,R3,#0
    ADD R3,R3,#12
; while (C > 0) {
W1    ADD R3,R3,#0
    BRnz ENDW1
; S = S + Mem[M];
    LDR R4,R2,#0
```

```
    ADD R1,R1,R4
; C = C - 1;
    ADD R3,R3,#-1
; M = M + 1;
    ADD R2,R2,#1
; }
    BR   W1
ENDW1 HALT
MINIT .fill x3100
.end
```

And Add Some Data

```
.orig x3100
.fill 3
.fill 4
.fill 5
.fill 1
.fill 6
.fill 2
.fill 7
.fill 8
.fill 9
.fill 12
.fill 10
.fill 11
.end
```

What Do We Get?

Memory

```
▶ 3000: x5260  
3001: x2409  
3002: x56E0  
3003: x16EC  
3004: x16E0  
3005: x0C06  
3006: x6880  
3007: x1244  
3008: x16FF  
3009: x14A1  
300A: x0FF9  
300B: x3100  
300C: xF025  
300D: x0000
```

Our Code

	AND R1, R1, #0
	LD R2, MINIT
	AND R3, R3, #0
	ADD R3, R3, #12
W1	ADD R3, R3, #0
	BRNZ ENDW1
	LDR R4, R2, #0
	ADD R1, R1, R4
	ADD R3, R3, #-1
	ADD R2, R2, #1
	BR W1
MINIT	ST R0, #-256
ENDW1	HALT
	NOP

Why the line?

What Do We Get? Page 2

Memory

30FF :	x0000
3100 :	x0003
3101 :	x0004
3102 :	x0005
3103 :	x0001
3104 :	x0006
3105 :	x0002
3106 :	x0007
3107 :	x0008
3108 :	x0009
3109 :	x000C
310A :	x000A
310B :	x000B
310C :	x0000

Our Code

NOP NOP

Question

How else does complex display the value x3100?

- A. NOP
- B. ADD R3,R3,#0
- C. BRNZ #‐6
- D. LDR R4,R2,#0
- E. ST R0,#‐256

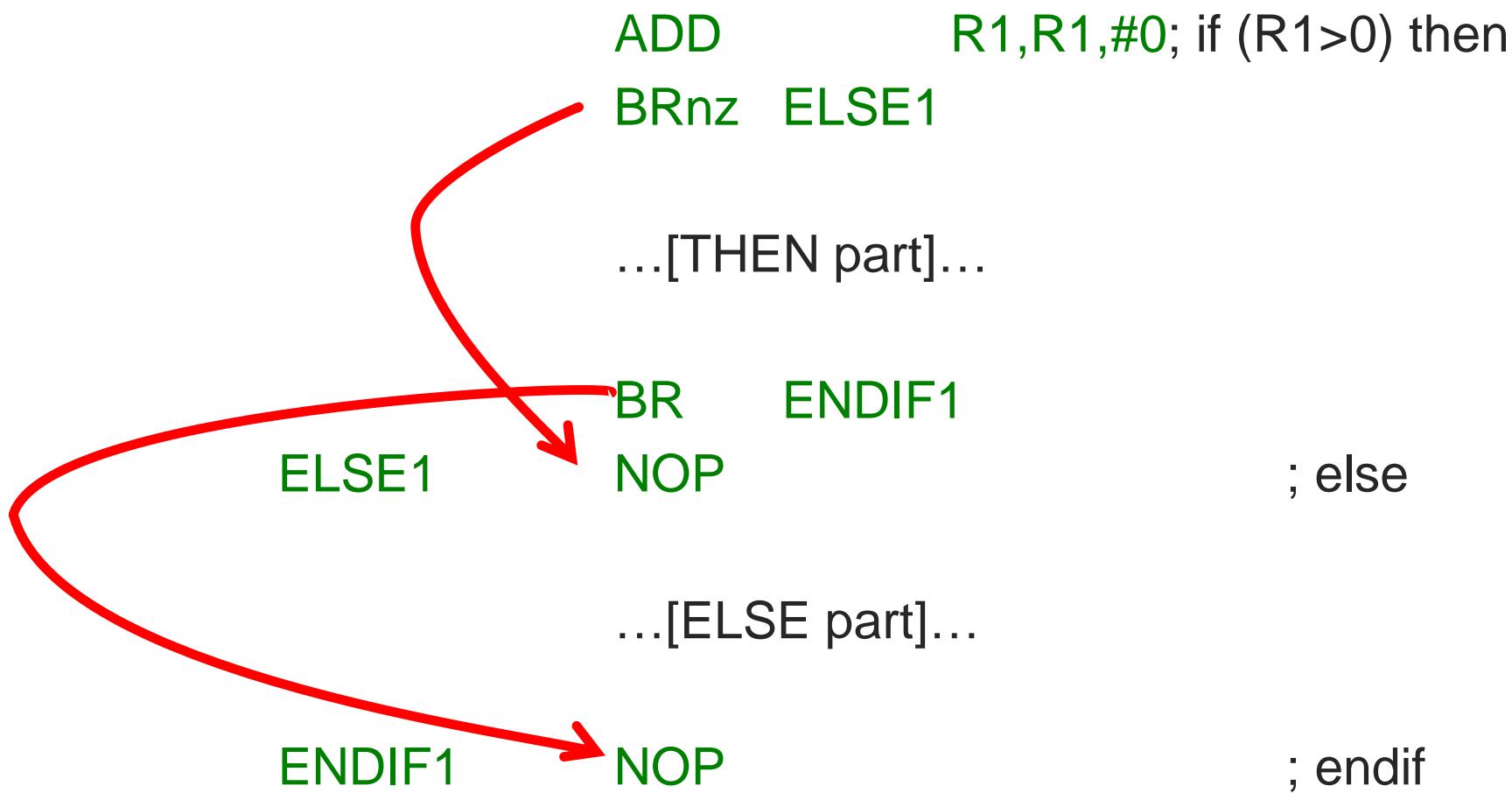


Today's number is 30,300

Code Templates

- ↗ Do you always have to write all the code from scratch?
- ↗ Do you think the machine code for **if/then** is very similar each time you write it?
- ↗ What about **while**, **for**, and **do**?
- ↗ So why shouldn't we make some templates instead of reinventing the wheel each time?
- ↗ Using template **will** make your life easier

if (R1>0) then .. else ..



for (init; R1>0; reinit)

FOR1

...[initialize loop]... ; for (init;

ADD R1,R1,#0

BRnz ENDF1

; R1>0;

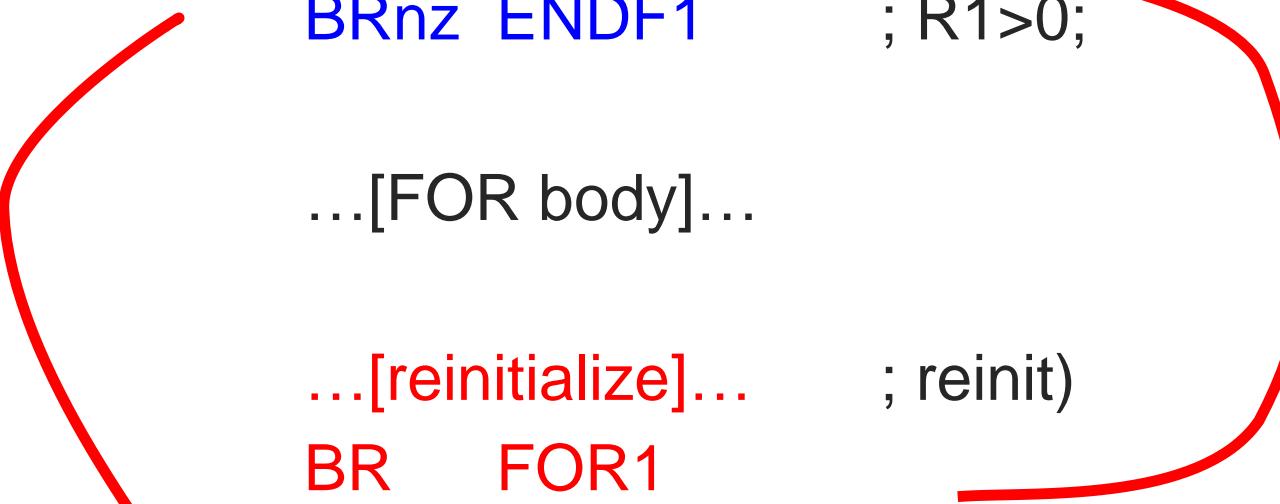
...[FOR body]...

...[reinitialize]... ; reinit)

BR FOR1

NOP

ENDF1



While ($R1 > 0$)

WHILE1

ADD R1,R1,#0

; while($R1 > 0$)

BRnz ENDW1

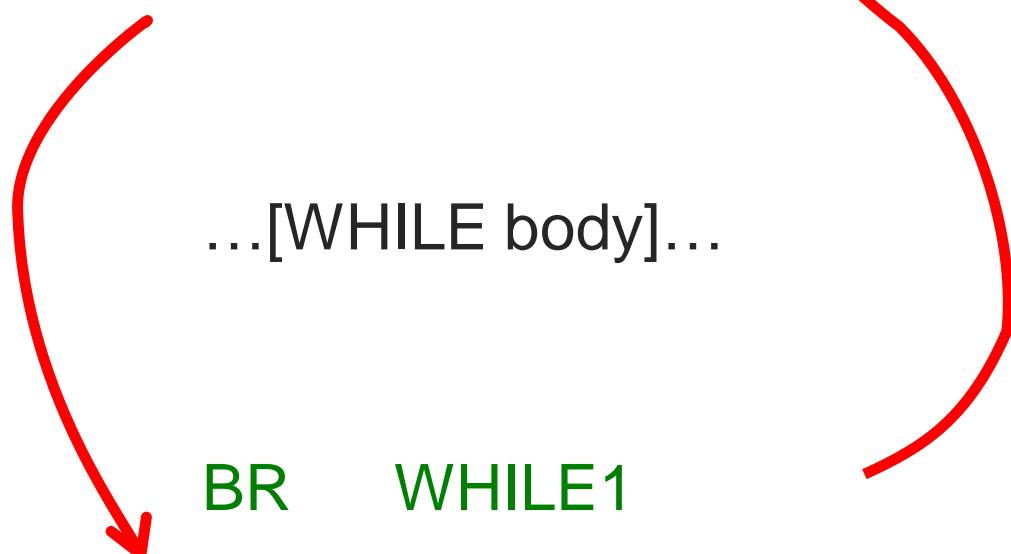
...[WHILE body]...

BR WHILE1

ENDW1

NOP

; endwhile



do ... while ($R1 > 0$);

DO1

NOP

; do

...[DO WHILE body]...

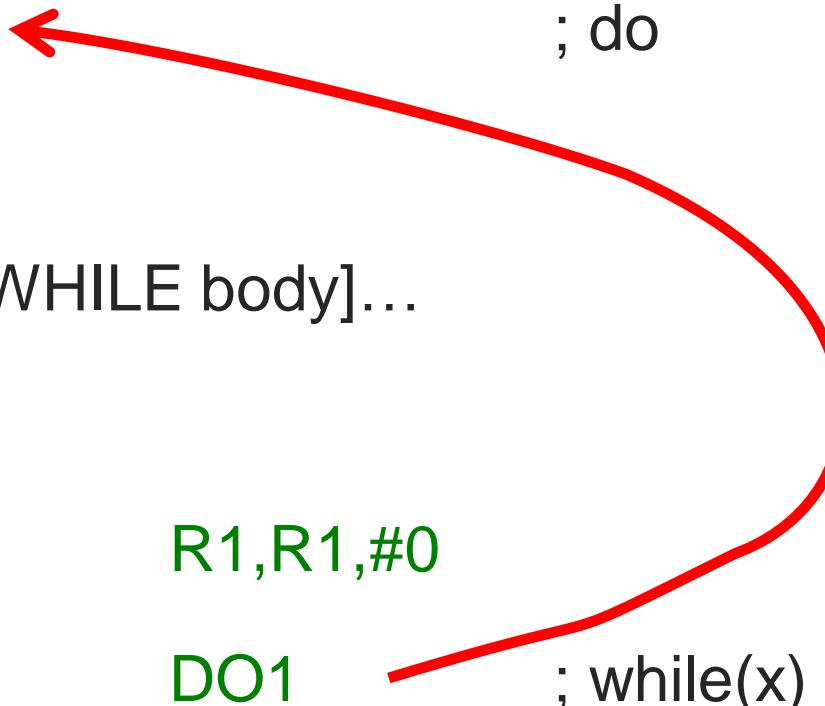
ADD

$R1, R1, #0$

BRp

DO1

; while(x)



if (A == 0 && B == 0)

; Go to ELSE part if either condition is false

LD R0,A
BRnp ELSE2
LD R0,B
BRnp ELSE2

; THEN code goes here

ELSE2 BR ENDIF2
 NOP

; ELSE code goes here

ENDIF2 NOP

Question

What is the **minimum number** of branch instructions required to implement the control structure “if ($A == 0$)” which includes no *else* part?

```
LDR    R0, A  
BRNP   Z  
; "then part"  
Z      NOP
```

- A. 0
- B. 1
- C. 2
- D. 3



if (A == 0 || B == 0)

; Trick: Negate the condition, swap THEN/ELSE parts
; & implement like && -- thanks Mr. DeMorgan

	LD R0,A
	BRz THEN3
	LD R0,B
	BRz THEN3
	;
	ELSE code goes here
THEN3	BR ENDIF3
	NOP
	;
	THEN code goes here
ENDIF3	NOP

Example

```
if (a < b) {  
    m = 5;  
}  
else {  
    m = a+1;  
}
```

Example: If – then – else

<pre>; if (a < b) { ; rewrite:if (a - b < 0) LD R1,a LD R2,b NOT R2,R2 ADD R2,R2,#1 ADD R1,R1,R2 BRZP EL1 ; m = 5; AND R1,R1,#0 ADD R1,R1,#5</pre>	<pre>ST R1,m BR EIF1 ;} else { EL1 NOP ; m = a+1; LD R1,a ADD R1,R1,#1 ST R1,m ;} EIF1 NOP</pre>
---	--

Example: For

```
for (l = 0; l < 10; l = l + 1) {  
    J = J + l;  
}
```

Example: For

<pre>;for (I = 0; I < 10; I = I + 1) { ; init: I = 0 AND R1,R1,#0 ST R1,I ; test (I - 10 < 0) FOR1 LD R1,I ADD R1,R1,#-10 BRzp EFOR1 ; J = J + I; LD R1,I</pre>	<pre>LD R2,J ADD R2,R2,R1 ST R2,J ; reinit I=I+1 LD R1,I ADD R1,R1,#1 ST R1,I }; BR FOR1 EFOR1 NOP</pre>
--	--

Example: Array Addressing

```
short a[5];
```

```
short q;
```

```
short i = 3;
```

```
q = a[2];
```

```
q = a[i];
```

```
a[i] = q;
```

```
q = a[i];
```

Example: Array addressing

; i is R3, q is R4

;short q;

;short a[5];

a .blkw 5

;short i = 3;

AND R3,R3,#0

ADD R3,R3,#3

;q = a[2];

LEA R0,a

LDR R4,R0,#2

;q = a[i];

LEA R0,a

ADD R0,R0,R3

LDR R4,R0,#0

;a[i] = q;

LEA R0,a

ADD R0,R0,R3

STR R4,R0,#0

;q = a[i];

LD R0,aaddr

ADD R0,R0,R3

LDR R4,R0,#0

...

aaddr .fill a

Addr Memory Word

a	a[0]
a+1	a[1]
a+2	a[2]
a+3	a[3]
a+4	a[4]

Example: For

```
short A[10];  
for (I = 0; I < 10; I = I + 1) {  
    J = J + A[I];  
}
```

Example: For

<pre>;short A[10]; A .blkw 10 ;for (I = 0; I < 10; I = I + 1) { ; init: I = 0 AND R1,R1,#0 ST R1,I ; test (I - 10 < 0) FOR1 LD R1,I ADD R1,R1,#-10 BRzp EFOR1 ; J = J + A[I]; LD R1,I LEA R2,A</pre>	<pre>ADD R1,R1,R2 LDR R1,R1,#0 LD R2,J ADD R2,R2,R1 ST R2,J ; reinit I=I+1 LD R1,I ADD R1,R1,#1 ST R1,I ; BR FOR1 EFOR1 NOP</pre>
---	--

Question

Implement the following statement in assembly language

$$K = M[J+1];$$

A. LDI R1,M
LD R2,J
ADD R1,R1,R2
ADD R1,R1,#1
LDR R1,R1,#0
ST R1,K

B. LEA R1,M
LD R2,J
ADD R1,R1,R2
LDR R1,R1,#0
ST R1,K

C. LD R1,M
LD R2,J
ADD R1,R1,R2
LDR R1,R1,#0
ST R1,K

D. LEA R1,M
LD R2,J
ADD R2,R2,#1
ADD R1,R1,R2
LDR R1,R1,#0
ST R1,K

R1	R2
Addr M	-
Addr M	Val J
Addr M	Val J+1
Addr M[J+1]	Val J+1
Val M[J+1]	Val J+1
Val M[J+1]	Val J+1

Templates for Indexing into an Array

Fetching ARRAY[I] (nearby)

```
LEA R1, ARRAY ;R1 is address of ARRAY[0]
LD R2, I ;R2 is index number
ADD R1, R1, R2 ;R1 is address of ARRAY[I]
LDR R1, R1, #0 ;R1 is value of ARRAY[I]
```

Storing ARRAY[I] (nearby)

```
LEA R1, ARRAY ;R1 is address of ARRAY[0]
LD R2, I ;R2 is index number
ADD R1, R1, R2 ;R1 is address of ARRAY[I]
STR R3, R1, #0 ;value of ARRAY[I] is R3
```

Fetching ARRAY[I] (far away)

```
LD R1, AD ;R1 is address of ARRAY[0]
LD R2, I ;R2 is index number
ADD R1, R1, R2 ;R1 is address of ARRAY[I]
LDR R1, R1, #0 ;R1 is value of ARRAY[I]
BR SK ;Don't execute the address
AD .fill ARRAY
SK NOP
```

Storing ARRAY[I] (far away)

```
LD R1, AD ;R1 is address of ARRAY[0]
LD R2, I ;R2 is index number
ADD R1, R1, R2 ;R1 is address of ARRAY[I]
STR R3, R1, #0 ;value of ARRAY[I] is R3
BR SK ;Don't execute the address
AD .fill ARRAY
SK NOP
```

Questions?

ADD ⁺	0001	DR	SR1	0	00	SR2
ADD ⁺	0001	DR	SR1	1		imm5
AND ⁺	0101	DR	SR1	0	00	SR2
AND ⁺	0101	DR	SR1	1		imm5
BR	0000	n	z	p		PCoffset9
JMP	1100	000		BaseR	000000	
JSR	0100	1			PCoffset11	
JSRR	0100	0	00	BaseR	000000	
LD ⁺	0010	DR			PCoffset9	
LDI ⁺	1010	DR			PCoffset9	

+ Indicates instructions that modify condition codes

LDR ⁺	0110	DR	BaseR	offset6
LEA	1110	DR		PCoffset9
NOT ⁺	1001	DR	SR	111111
RET	1100	000	111	000000
RTI	1000		000000000000	
ST	0011	SR		PCoffset9
STI	1011	SR		PCoffset9
STR	0111	SR	BaseR	offset6
TRAP	1111	0000		trapvect8
reserved	1101			

+ Indicates instructions that modify condition codes

The Stack and Subroutines

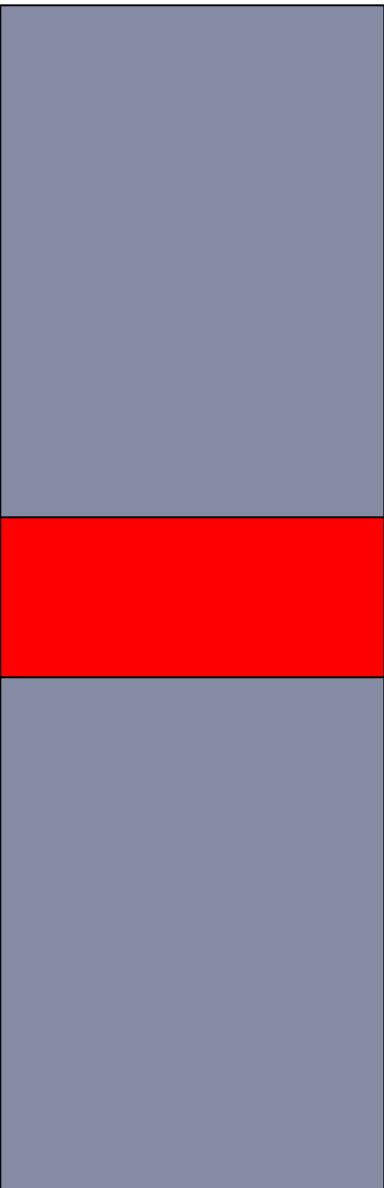


- Subroutines - overview
- The Stack
- LC-3 calling convention
 - Stack Pointer and Frame Pointer
 - Caller and Callee
 - Examples

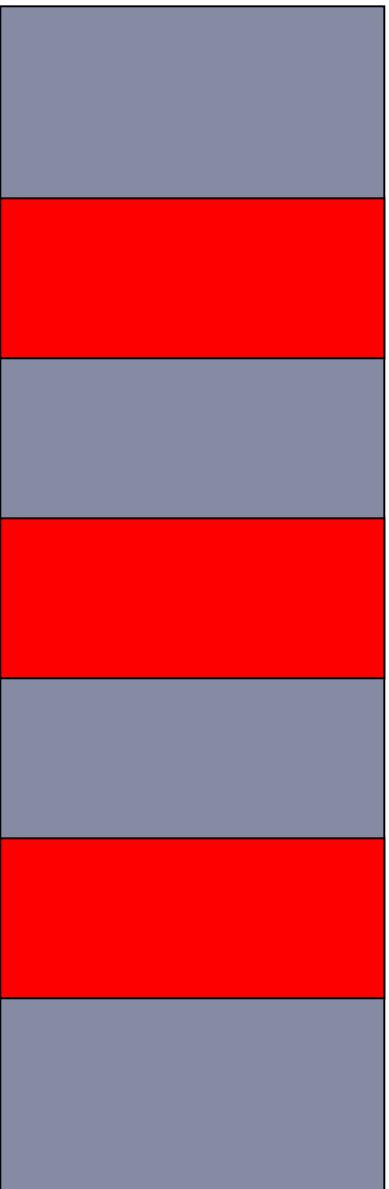
Why do we have subroutines?

- All of the following are related to subroutines:
 - Methods
 - Functions
 - Procedures
- We call them “subroutines” in assembly.
- Fundamentally, subroutines are about reusing code

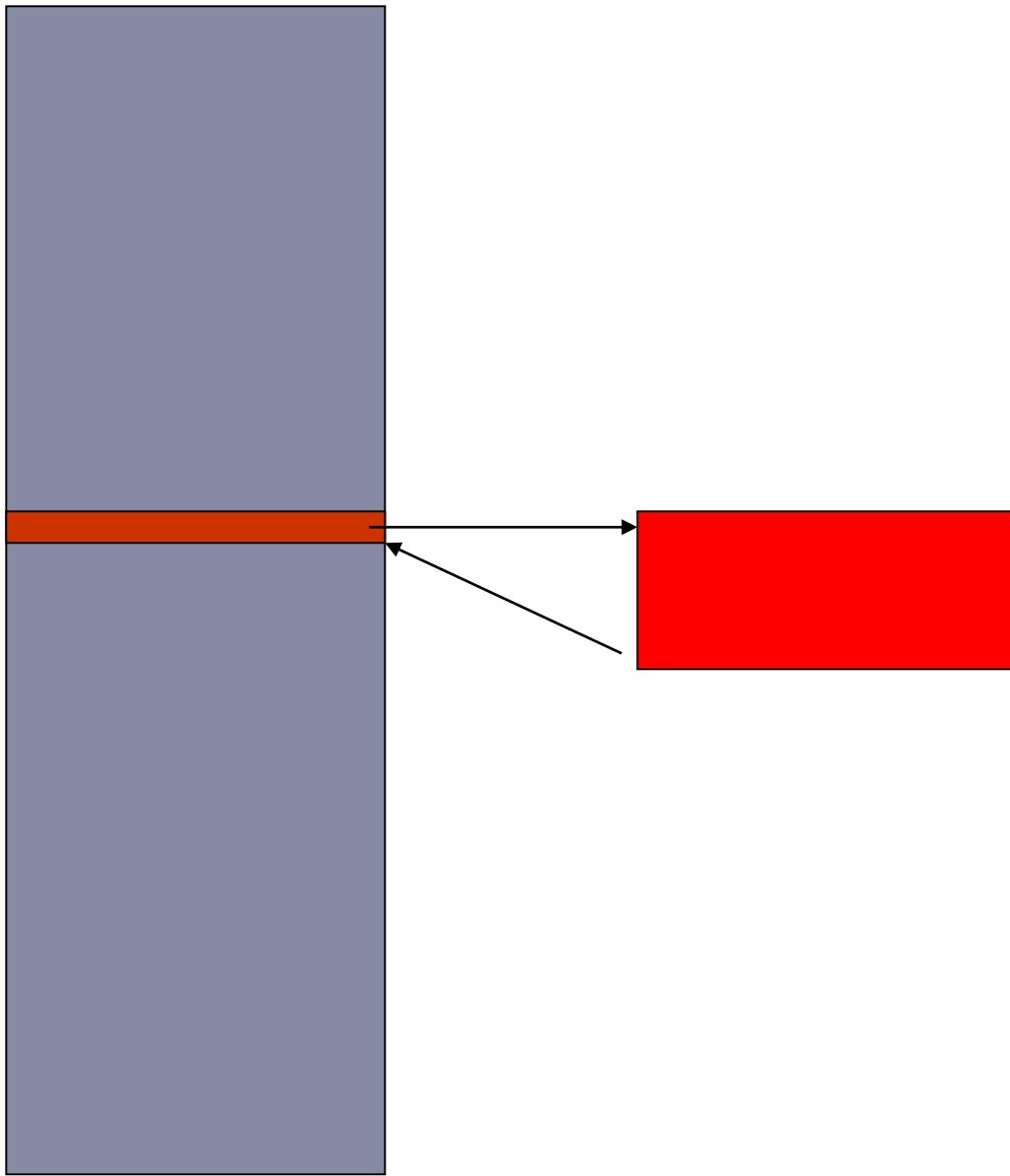
Code That We Might Want to Reuse



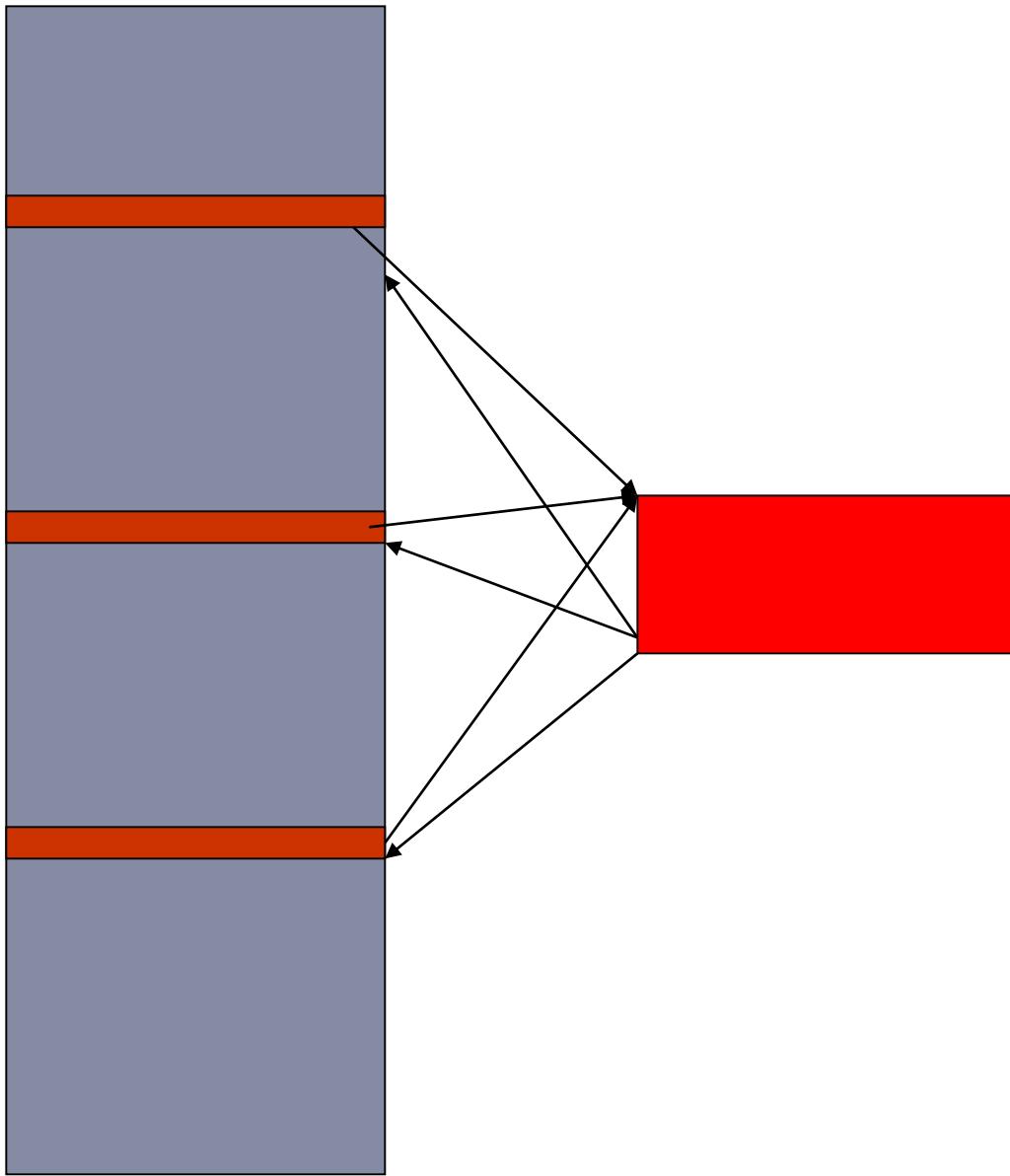
Reusing Code by Repeating



Turn Into a Subroutine



Using A Subroutine Call



Call/Return Mechanism

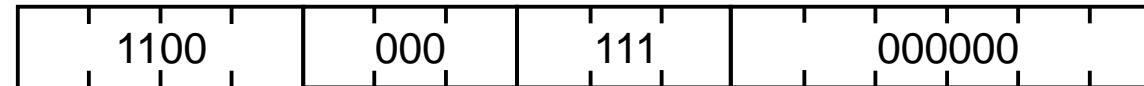
JSR



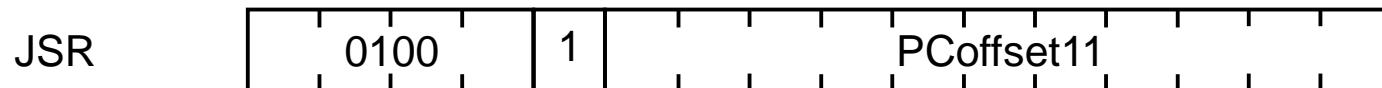
JSRR



RET



What does JSR do?



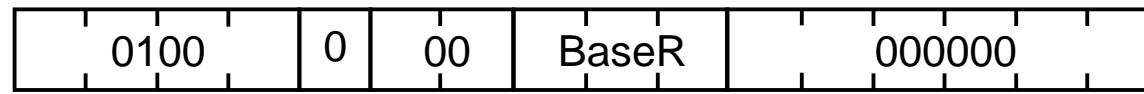
$R7 \leftarrow PC$

$PC \leftarrow PC + PCOffset11$

(remember, PC has been incremented in the Fetch cycle)

What does JSRR do?

JSRR

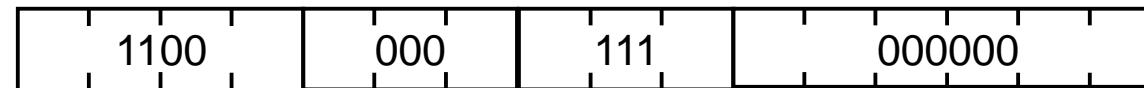


$R7 \leftarrow PC$

$PC \leftarrow BaseR$

What does RET do?

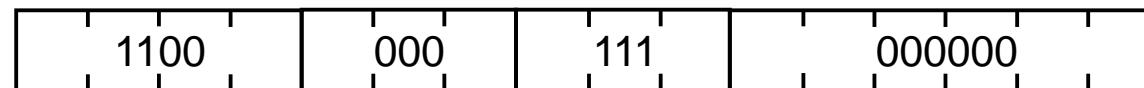
RET



What does RET do?

Jumps to R7

RET



Short and Long Reach Subroutines

```
jsr sub
```

```
sub    xxx xxx xxx
```

```
...
```

```
...
```

```
...
```

```
ret
```

```
ld R2, far  
jsrr r2
```

```
far    .fill sub
```

```
...
```

```
...
```

```
sub    xxx xxx xxx
```

```
ret
```

Question

The RET instruction has the same opcode as

- A. JSR
- B. JSRR
- C. JMP 
- D. BR

How do we write a subroutine?

- What issues do we need to consider?

Issue 1: What if our subroutine calls another subroutine?

- ↗ What is R7 used for?
- ↗ What happens to R7 if our subroutine calls another subroutine?

Preserve the Return Address (R7)

- We'd better find a way for our subroutine to save its R7 return address somewhere.
- That way, we are safe to call another subroutine.
- Then, we can restore our original R7, and know where to return to when we are done.

Issue 2: we only have a few registers

- With only 8 registers (R0-R7)
 - We are going to quickly run out of registers to use.
- Our main program is using some registers
 - Our subroutine also needs to use some registers
 - So let's save the old values from the registers we'll use
 - And then we can "borrow" these registers to use our subroutine
 - When we're done, we'll put back the original values into those registers – like they were never changed!

Saving Registers in a Subroutine

```
; Subroutine that needs to use r1, r2 and r3
subr      st r1, r1save
           st r2, r2save
           st r3, r3save
;
           work is done here
           ld r1, r1save
           ld r2, r2save
           ld r3, r3save
           ret

r1save   .fill 0
r2save   .fill 0
r3save   .fill 0
```

See any problems?

- ↗ How much memory space is lost if every subroutine declares space to save registers?
- ↗ What happens if our subroutine calls *itself*?
 - ↗ Recursion
 - ↗ We'll step on the saved registers in memory, and lose those values.

Make It Bulletproof!

- ↗ What do we need to save?
 - ↗ Old values in the registers we will use
 - ↗ Our return address (from R7)
- ↗ Where should we save these values?

How do we do it?

- ↗ We can't always save things in the same place
 - ↗ E.g. a fixed memory location
- ↗ So what kind of data structure might we use?
 - ↗ Answer: a stack

What is a Stack?

- ↗ Stack
 - ↗ Last in/First Out
 - ↗ Supports two operations: Push/Pop

How do we implement the stack?

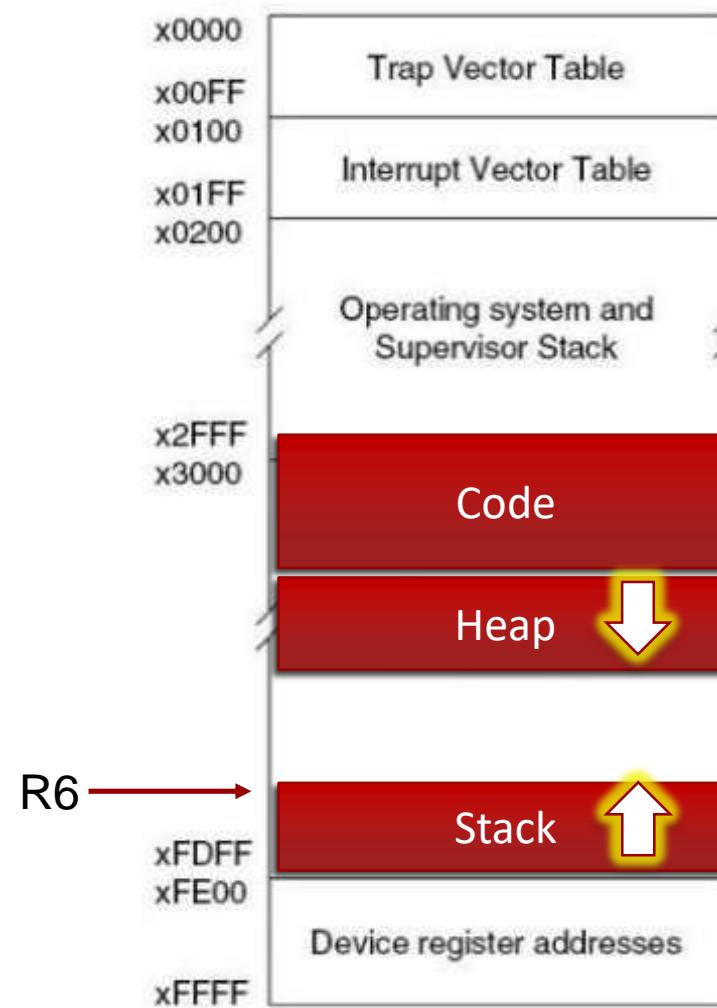
- ↗ Stack is located in some designated area of memory
- ↗ We need to store the address of the top of the stack somewhere.
- ↗ The stack could "grow" in either direction (up or down)

Top of the stack

- We only need to keep track of one thing:
 - The “top” of the stack (TOS)
 - The memory location where the last value was pushed onto the stack
 - Which is also the next thing to pop off the stack
- Let’s use R6 to store the “stack pointer”
 - The top of the stack

Where Do We Put the Stack?

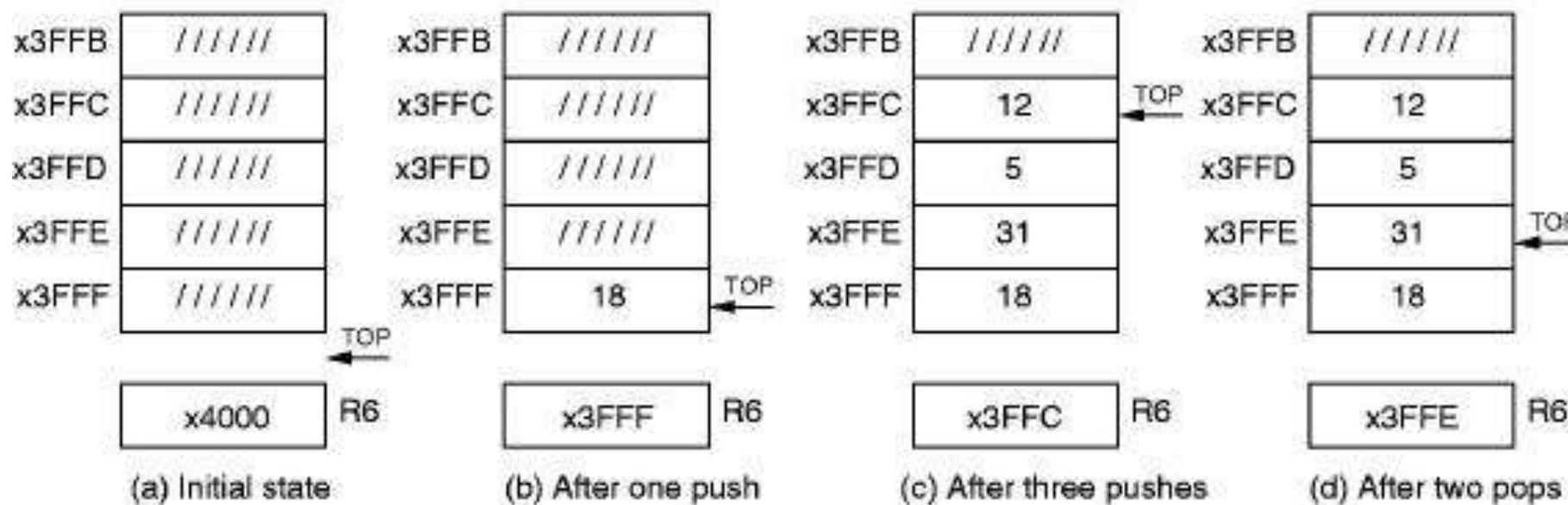
- Turns out there's a convenient spot for the stack on the LC-3
- Since the heap* grows up, let the stack grow down to lower memory addresses!
- Note the memory address are listed low to high on the page; down is high to low, so the arrow appears to point up!



* We'll tell you what the heap is later, when we get into C programming.

A Software Stack Convention

- ↗ Implemented in memory
 - ↗ The Top Of Stack moves as new data is entered
 - ↗ Here R6 is the Stack Pointer, a pointer to the Top Of Stack
 - ↗ Note that “down” is again “up” on this page...



Push & Pop

↗ Push

- ↗ Decrement stack pointer (our stack is growing *down to lower memory addresses*)
- ↗ Then write data in R0 to new TOS

```
PUSH    ADD     R6, R6, # -1  
        STR     R0, R6, # 0
```

↗ Pop

- ↗ Read data at current TOS into R0
- ↗ Then increment stack pointer

```
POP     LDR     R0, R6, # 0  
        ADD     R6, R6, # 1
```

Running out of memory

- ↗ What if stack is already full or empty?
 - ↗ Before pushing, we could test for overflow
 - ↗ Before popping, we could test for underflow
- ↗ In this class, we'll ignore bounds checking
 - ↗ In the real world, it definitely needs to happen
- ↗ What happens when the stack pointer runs into the heap?
 - ↗ What do you call that?
 - ↗ Hint: a *popular website*

Question

How do we push a word in R0 onto the LC-3 stack?

- A. Decrement R7
Store R0 at the address in R7
- B. Store R0 at the address in R6
Decrement R6
- C. Store R0 at the address in R7
Increment R7
- D. Decrement R6
Store R0 at the address in R6



How does a stack help with subroutines?

- ↗ We can push R7 onto the stack – to save the return address
- ↗ We can push each of the registers onto the stack, to save copies of their old values
 - ↗ Then we can borrow those registers to re-use inside of our subroutine
- ↗ When our subroutine is done, we just pop the return address and original registers back off the stack

What else might a subroutine need to know, or store?

- ↗ We're already going to save on the stack:
 - ↗ R7 (return address)
 - ↗ Copies of registers
- ↗ What other data does a subroutine use or need?
- ↗ What else might we want to store on the stack?
 - ↗ Arguments
 - ↗ Return Value
 - ↗ Local variables

Stack Frame

- All of the data on the stack, relevant to our subroutine
 - Arguments
 - Return Value
 - Local Variables
 - Saved Return Address (R7)
 - Saved Registers

A subroutine stack frame

```
int foo(int x, int y, int z){
```

```
    int temp;
```

3 arguments

```
    temp = x + y + z;
```

1 local variable

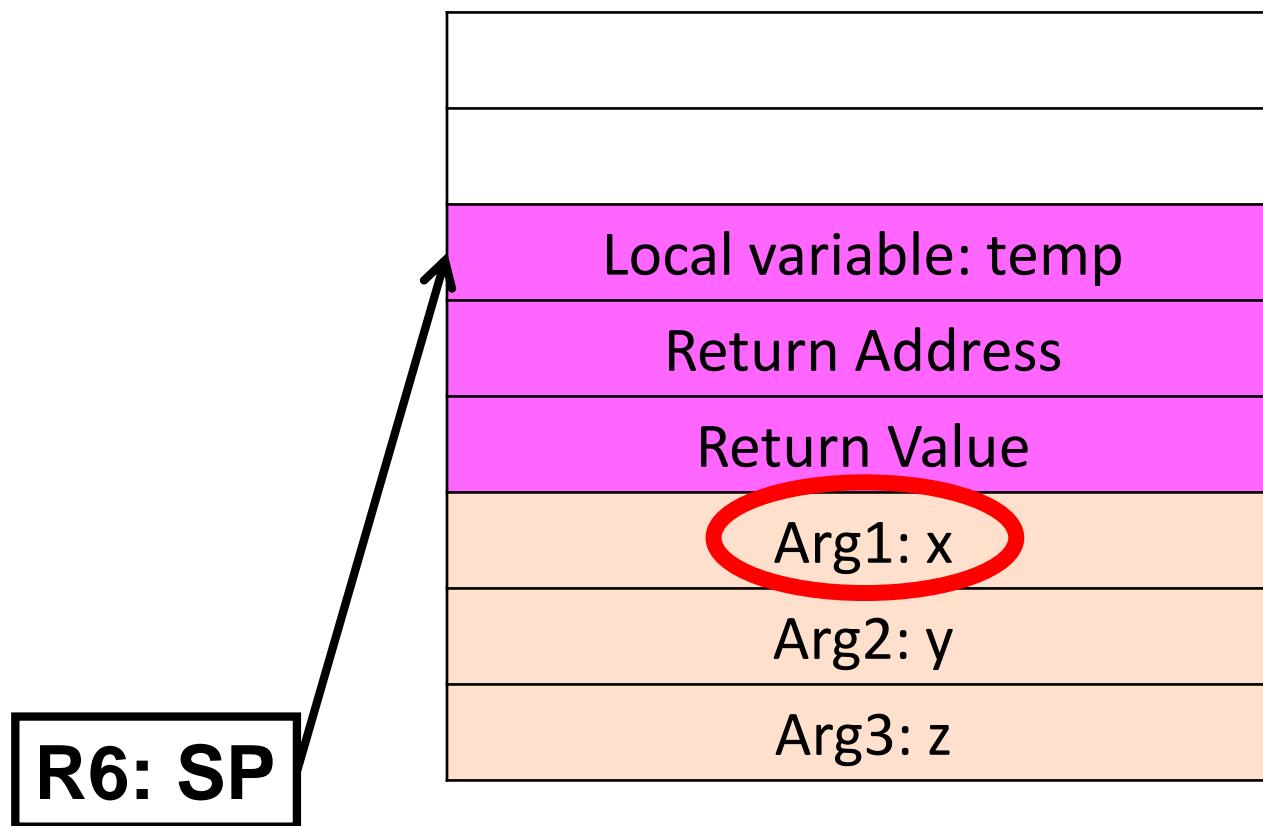
```
    return temp;
```

return value

```
}
```

saved return address (R7)

Stack Frame



Where is x?

At SP+3

FFFF

WAIT!

- ↗ Are we always sure where x is?
- ↗ Is it always at R6+3?
 - ↗ R6 is the stack pointer (SP)

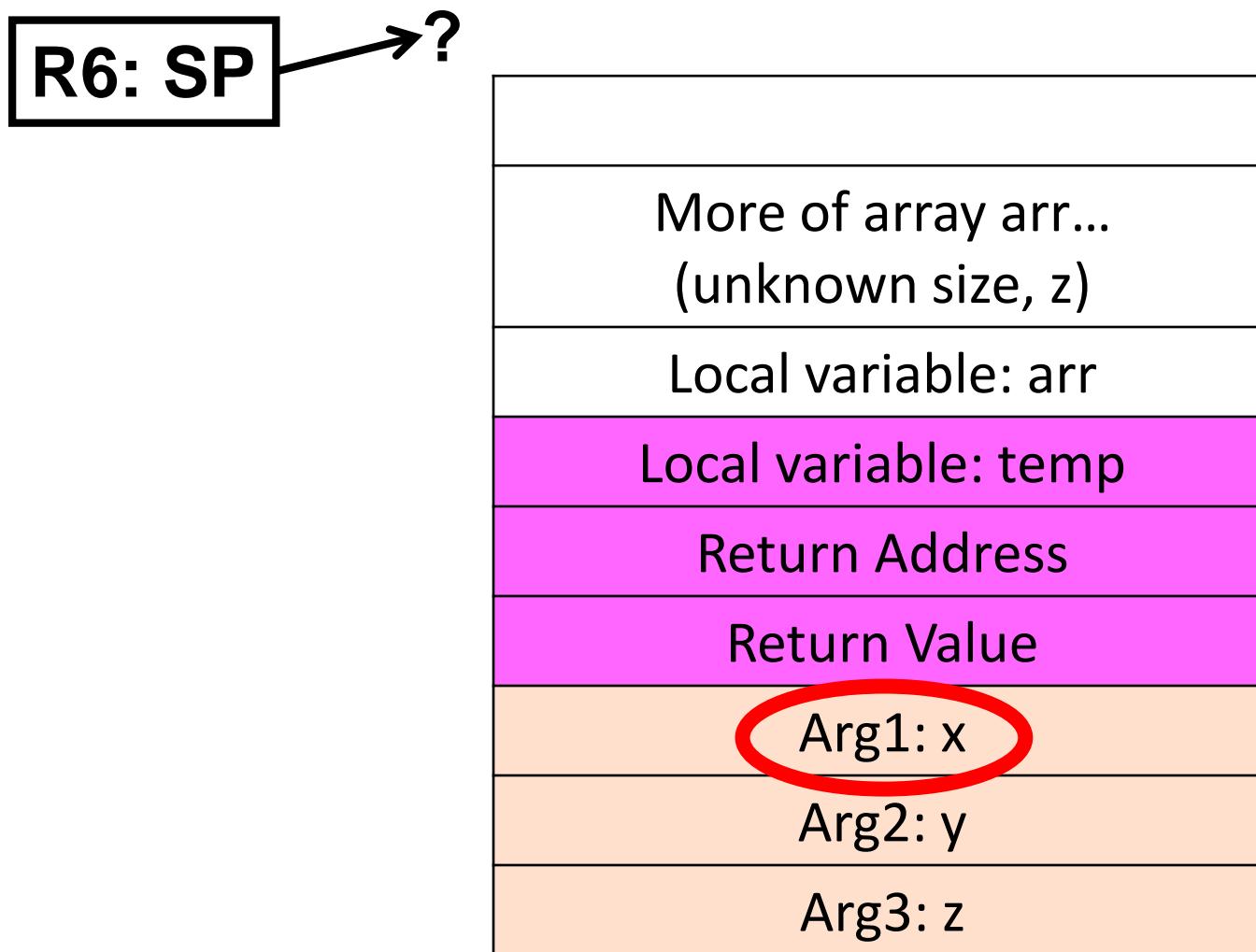
What If foo Looked Like This?

```
int foo(int x, int y, int z) {  
    int temp;  
  
    int arr[z];  
  
    ...  
  
    x = 1;  
  
    ...  
  
}
```

What If foo Looked Like This?

```
int foo(int x, int y, int z) {  
    int temp;  
  
    int arr[z];  
  
    ...  
  
    x = 1;  
  
    ...  
  
}
```

The New Picture



0000

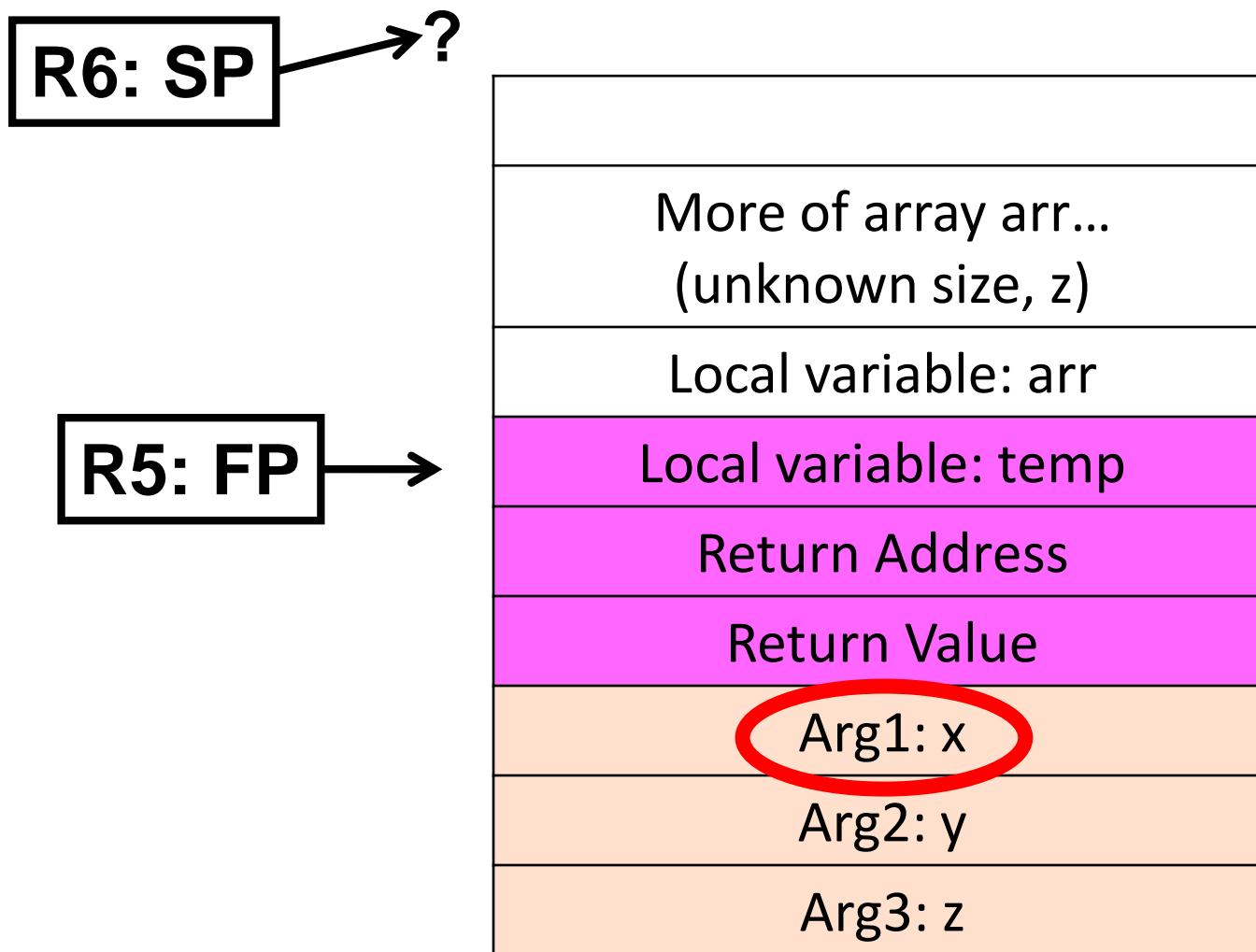
Where is x now?
(relative to R6)

FFFF

Solution: Frame Pointer!

- Keep a “known” location in the stack frame in another register
- Which register? R5
- We’re going to **ALWAYS** point it to the first local variable
 - (and we’ll always reserve space for at least one local var, even if we don’t have any local variables)
- **The frame pointer, R5, is an anchor**
 - At a predictable location in the stack frame

The Frame Pointer, R5



0000

Where is x now?

(relative to R5)

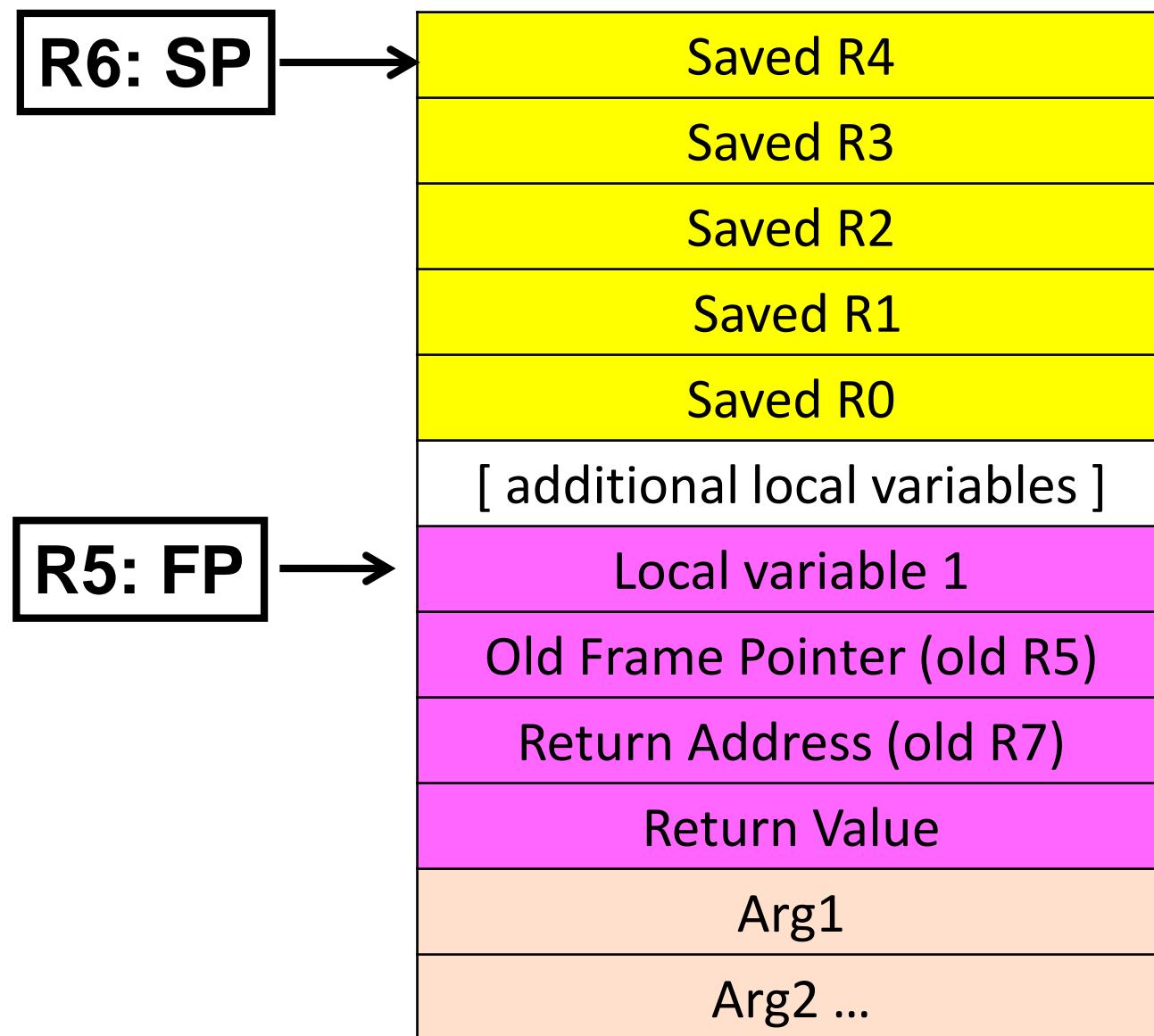
Always at a fixed
offset after R5

FFFF

Special Purpose Registers

- ↗ R0
 - ↗ R1
 - ↗ R2
 - ↗ R3
 - ↗ R4
 - ↗ R5 – Frame Pointer
 - ↗ R6 – Stack Pointer
 - ↗ R7 – Return Address
-
- ↗ **From now on (with subroutines),**
 - ↗ **Only use R0-R4 as general purpose registers!**

LC-3 Stack Frame



Note that we will also save the old R5 Frame Pointer (from the caller)

How do we organize the stack frame – for any subroutine?

↗ Adopt a Calling Convention

Fair Warning!

- ↗ There are zillions of ways to write a calling convention
- ↗ Calling conventions are intimately connected to an ISA
- ↗ Each is difficult to get “right”.
- ↗ Patt, et al. have picked one and have done it “right”.
- ↗ If you try to re-invent it, you will make yourself a lot of extra trouble.
- ↗ **Use the Patt LC-3 calling convention we'll teach you.**

- The **stack frame**, also known as an **activation record**, is the collection of all data on the stack associated with one subprogram call.
- The stack frame generally includes the following components:
 - The return address
 - Argument variables passed on the stack
 - Local variables
 - Saved copies of any registers modified by the subroutine that need to be restored

↗ Caller

- ↗ The code that will call a subroutine, e.g.
- ↗ `m = mult(a,b);`

↗ Callee

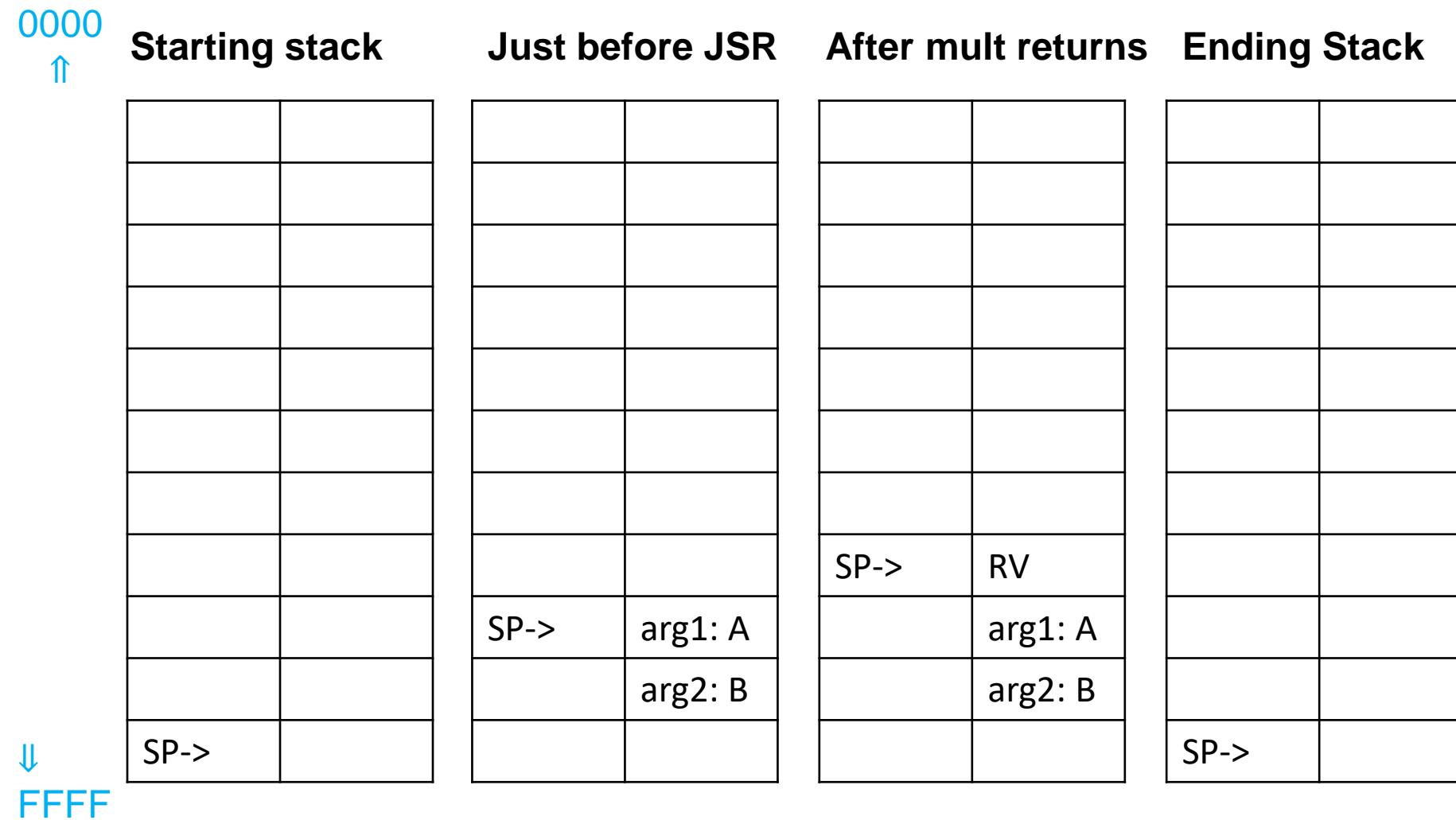
- ↗ The subroutine definition

```
int mult(a,b) {  
    return a*b  
}
```

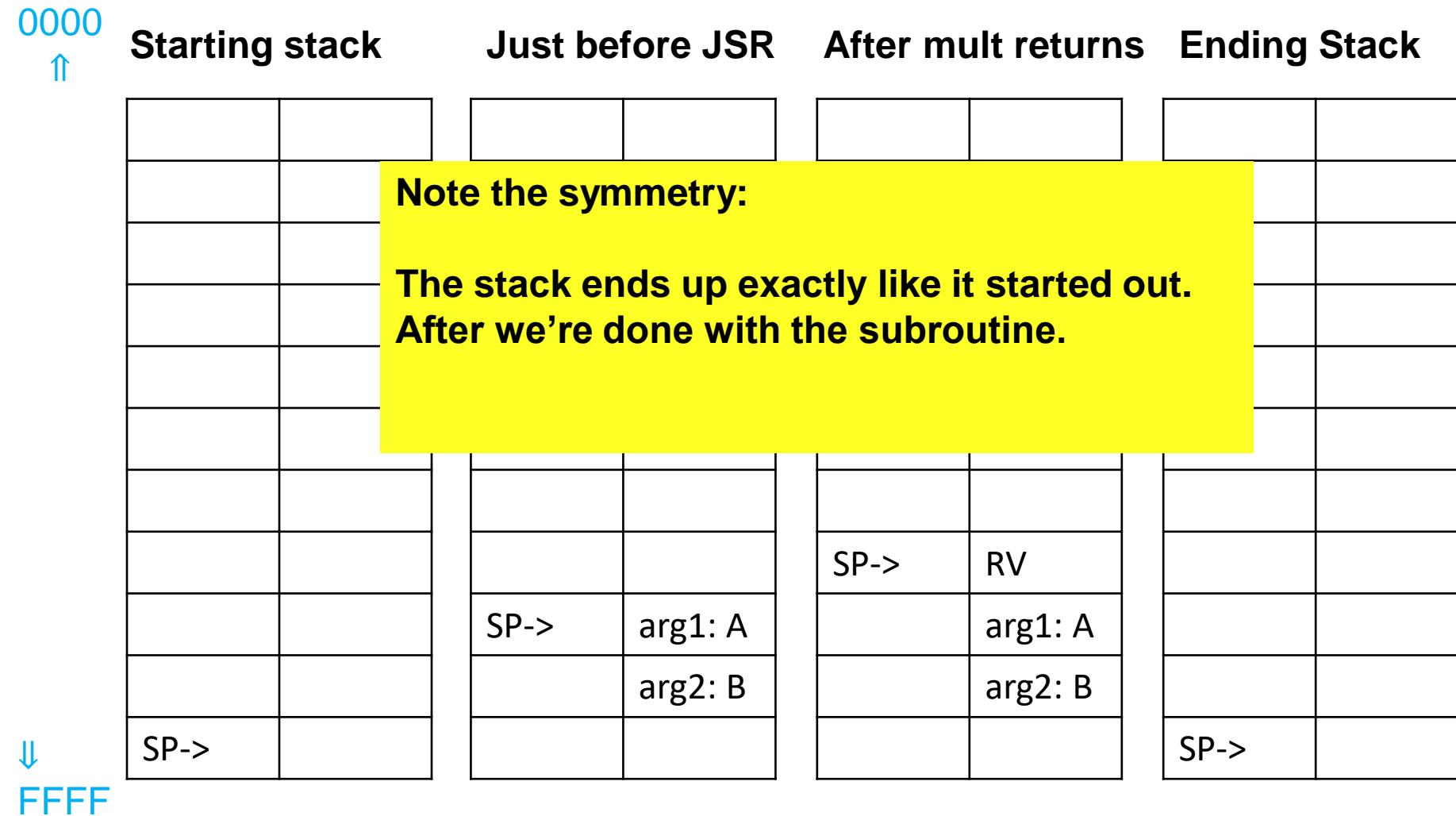
- What do we do to call a subroutine?
 - Push arguments (in reverse order)
 - JSR

- What do we do after the subroutine returns?
 - Pop return value
 - Pop the arguments

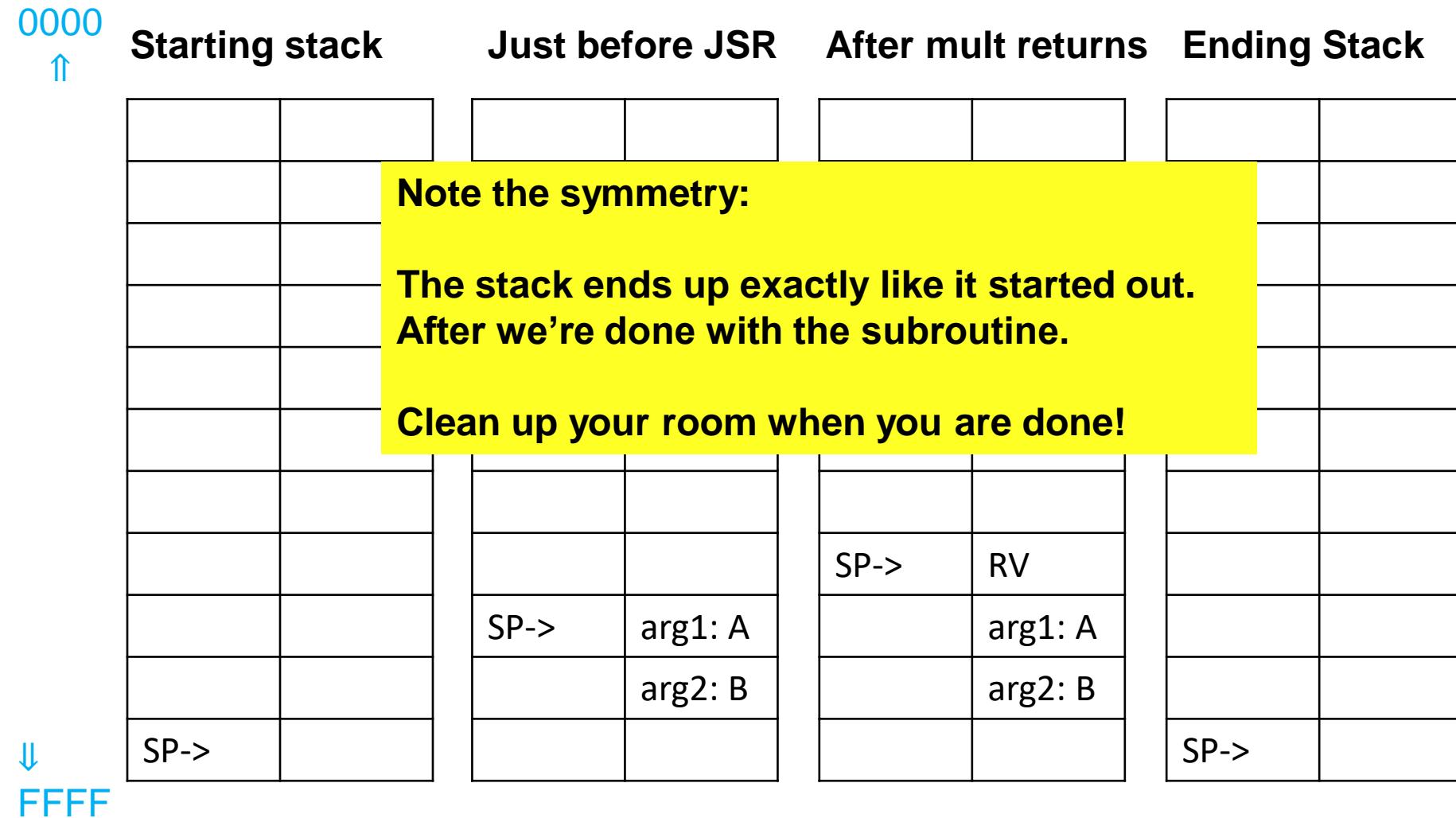
Caller: mult(A,B)



Caller: mult(A,B)



Caller: mult(A,B)



Question

At the moment a subroutine returns, what can we say about the stack?

- A. The arguments have been popped off the stack
- B. The stack pointer has the same value as it did before the call
- C. The stack pointer is one less than its value before the call
- D. The stack pointer contains the return value of the subroutine



What the effect of JSR?

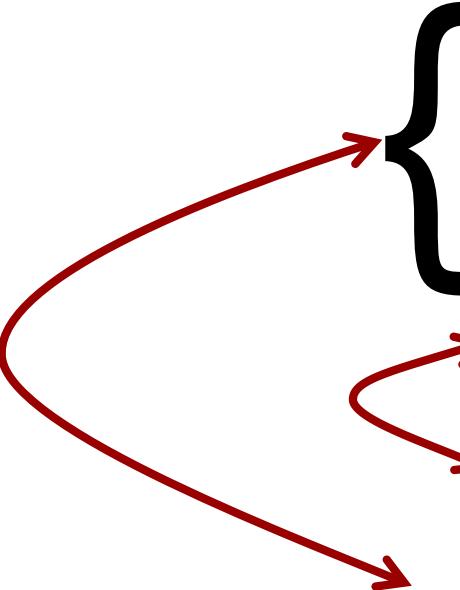
- ↗ A subroutine effectively pushes its return value on the stack
- ↗ That is what JSR looks like
 - ↗ From the perspective of the caller

Assembly: Caller of mult(a,b)

```
; let's call mult          ; after MULT returns
; M = mult(A,B);           ; pop the return value
; assume M, A, B, MULT are LDR R1, R6, #0
;   labels                 ADD R6, R6, #1
;                         ; save the ret val in M
; push arguments           ST R1, M
;   in reverse order       ; pop the two args, A and B
; push B                   ADD R6, R6, #2
LD R1, B
ADD R6, R6, #-1
STR R1, R6, #0
; push A
LD R1, A
ADD R6, R6, #-1
STR R1, R6, #0
; call mult
JSR MULT
```

Symmetry of Caller (stack frame)

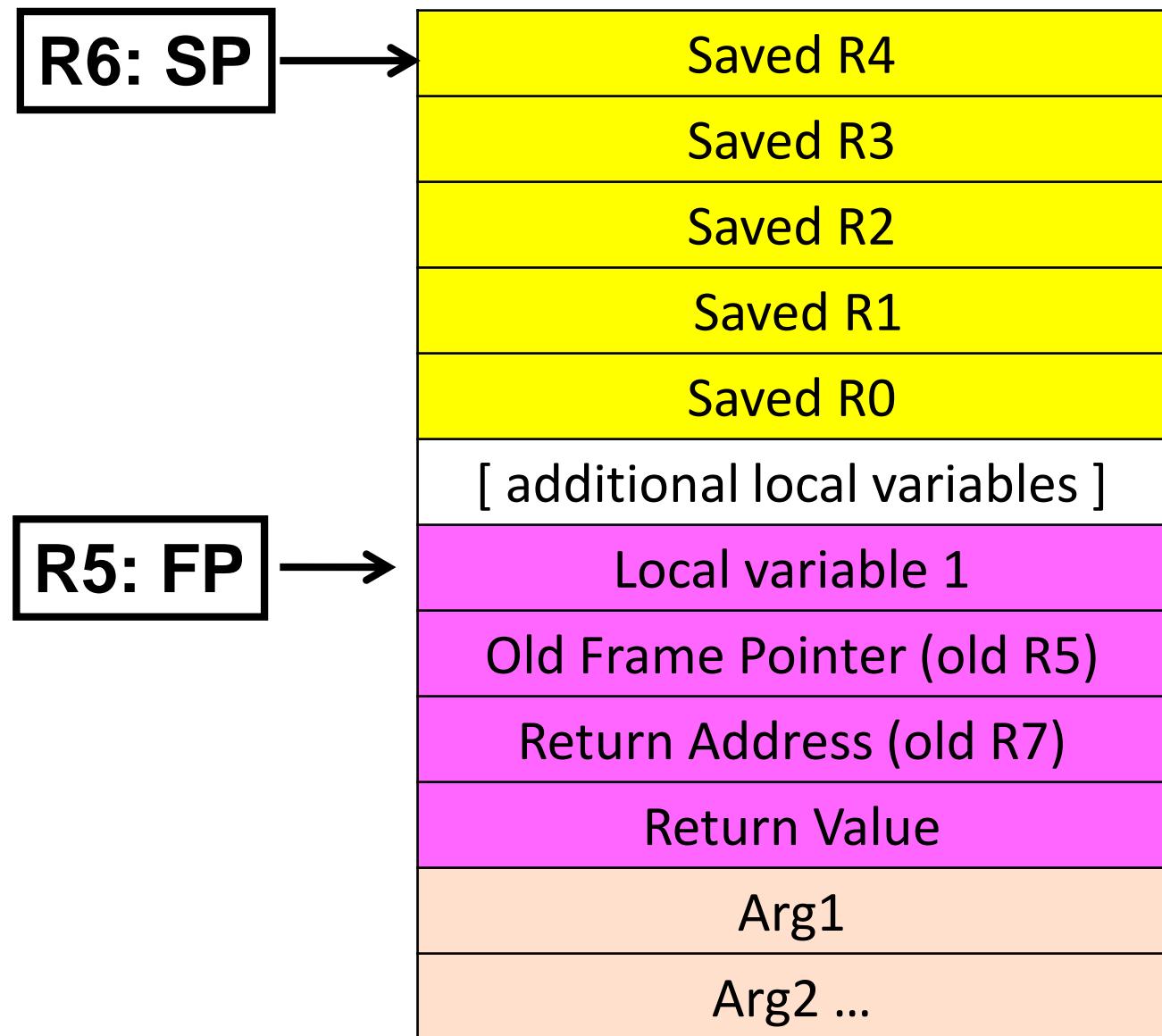
```
; m = mult(a,b)          ; caller's perspective
LD    R0, B              ; push B
ADD   R6, R6, -1
STR   R0, R6, 0
LD    R0, A              ; push A
ADD   R6, R6, -1
STR   R0, R6, 0
JSR   MULT               ; call subroutine (e.g. push ret val)
LDR   R0, R6, 0           ; pop return value
ADD   R6, R6, 1
ST    R0, M               ; save return value at m
ADD   R6, R6, 2           ; pop two arguments (A,B)
```



Stack Frame - Callee

- When a subroutine is called (**stack buildup**)
 - Save some registers (callee-saved)
 - Make room for local variables
- Do the work of the subroutine
- When procedure is about to return (**stack teardown**)
 - Prepare return value
 - Restore registers
 - RET

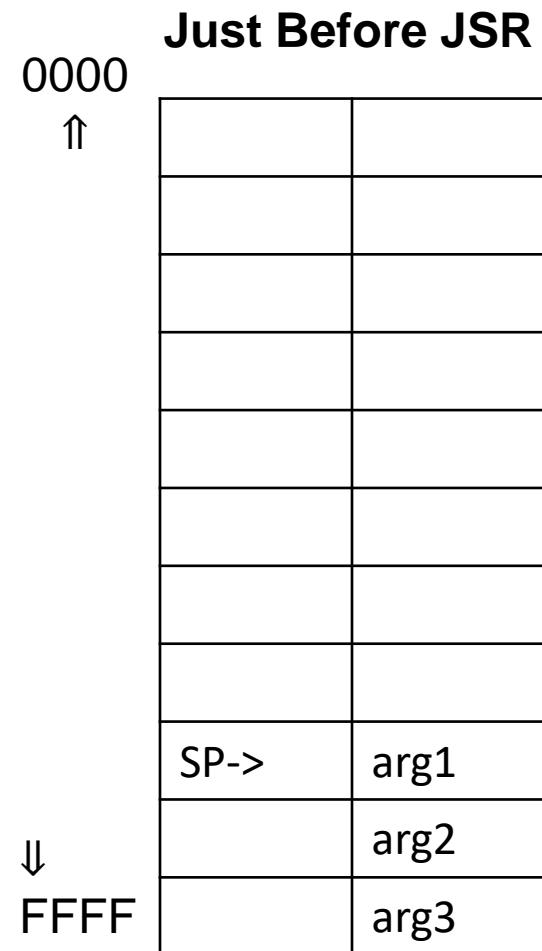
LC-3 Stack Frame



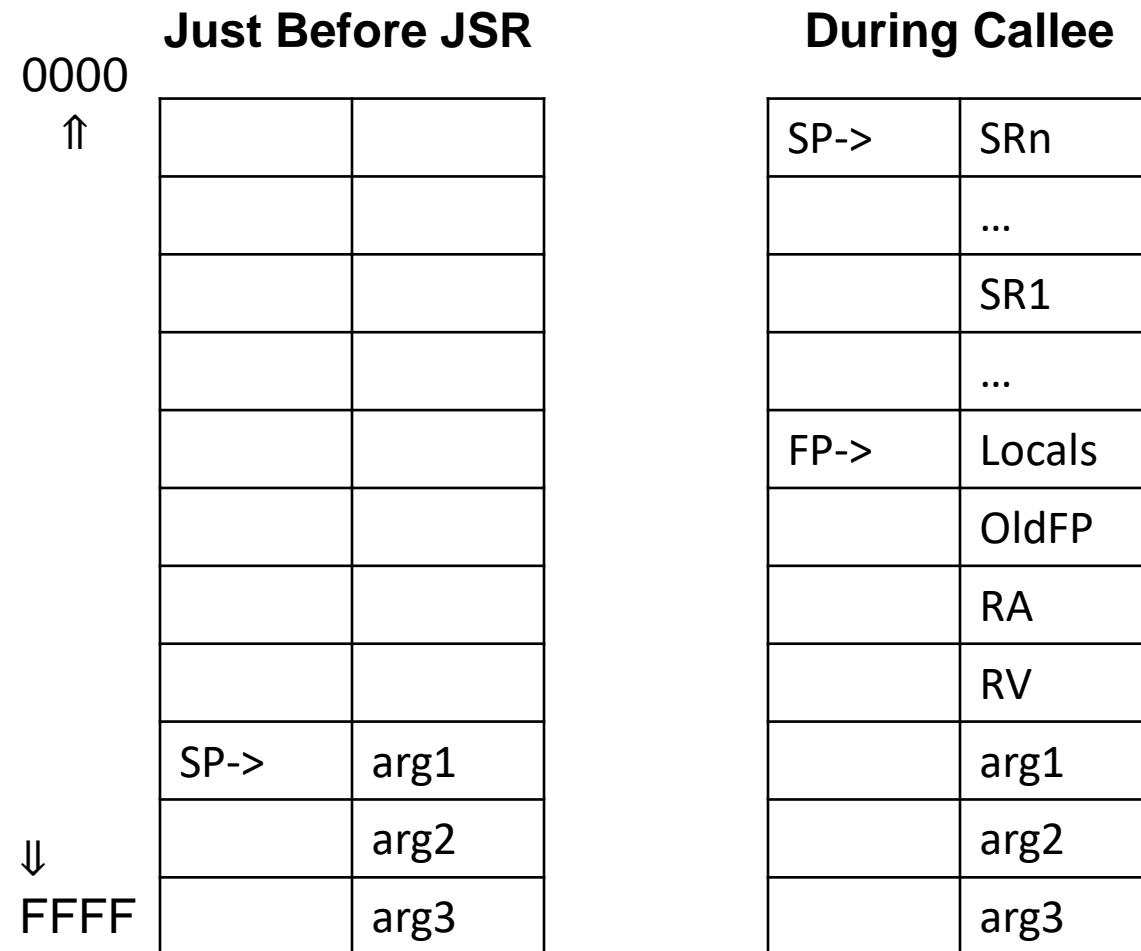
0000

FFFF

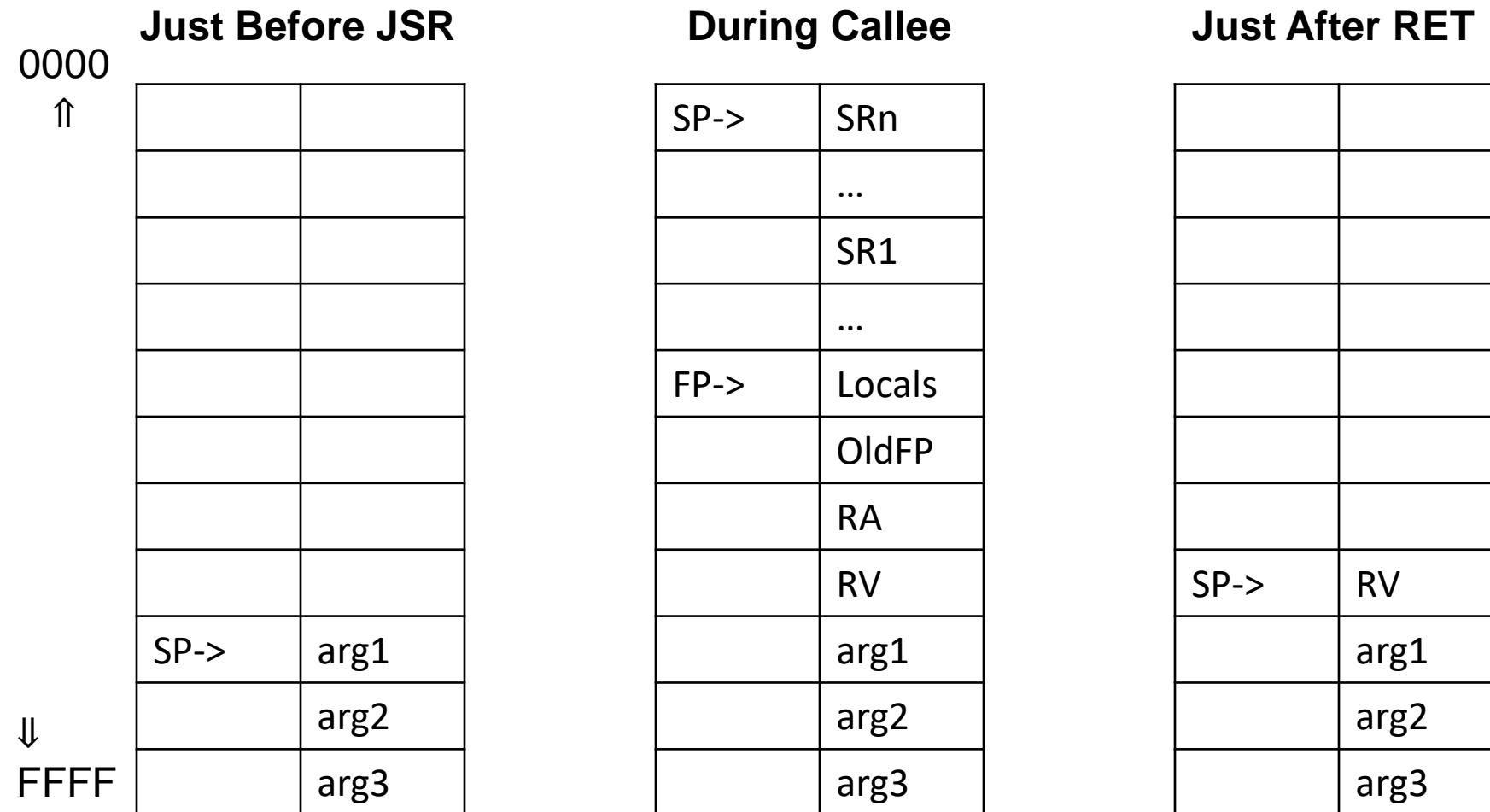
After Callee Teardown



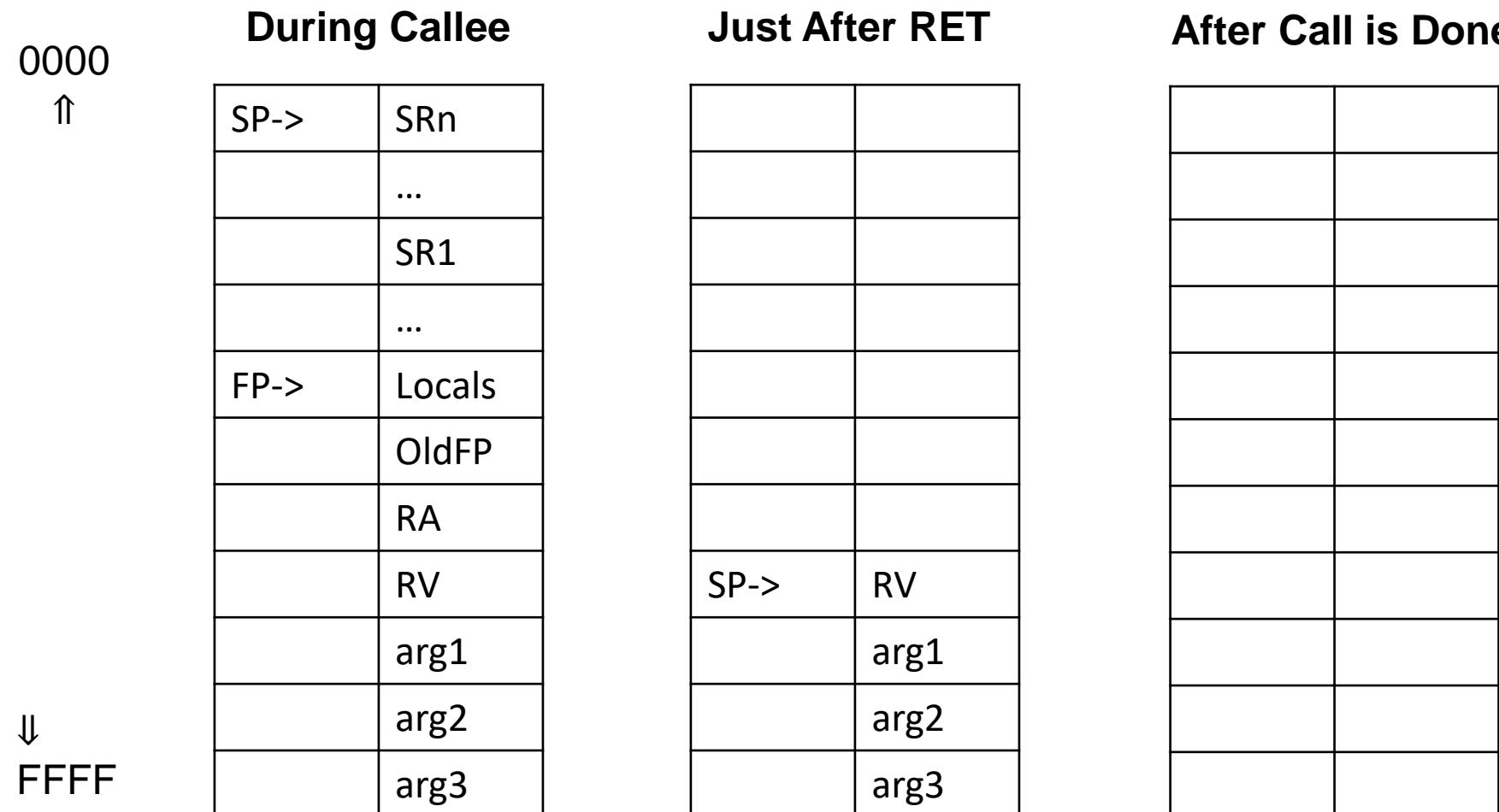
After Callee Teardown



Stack Frame Progression



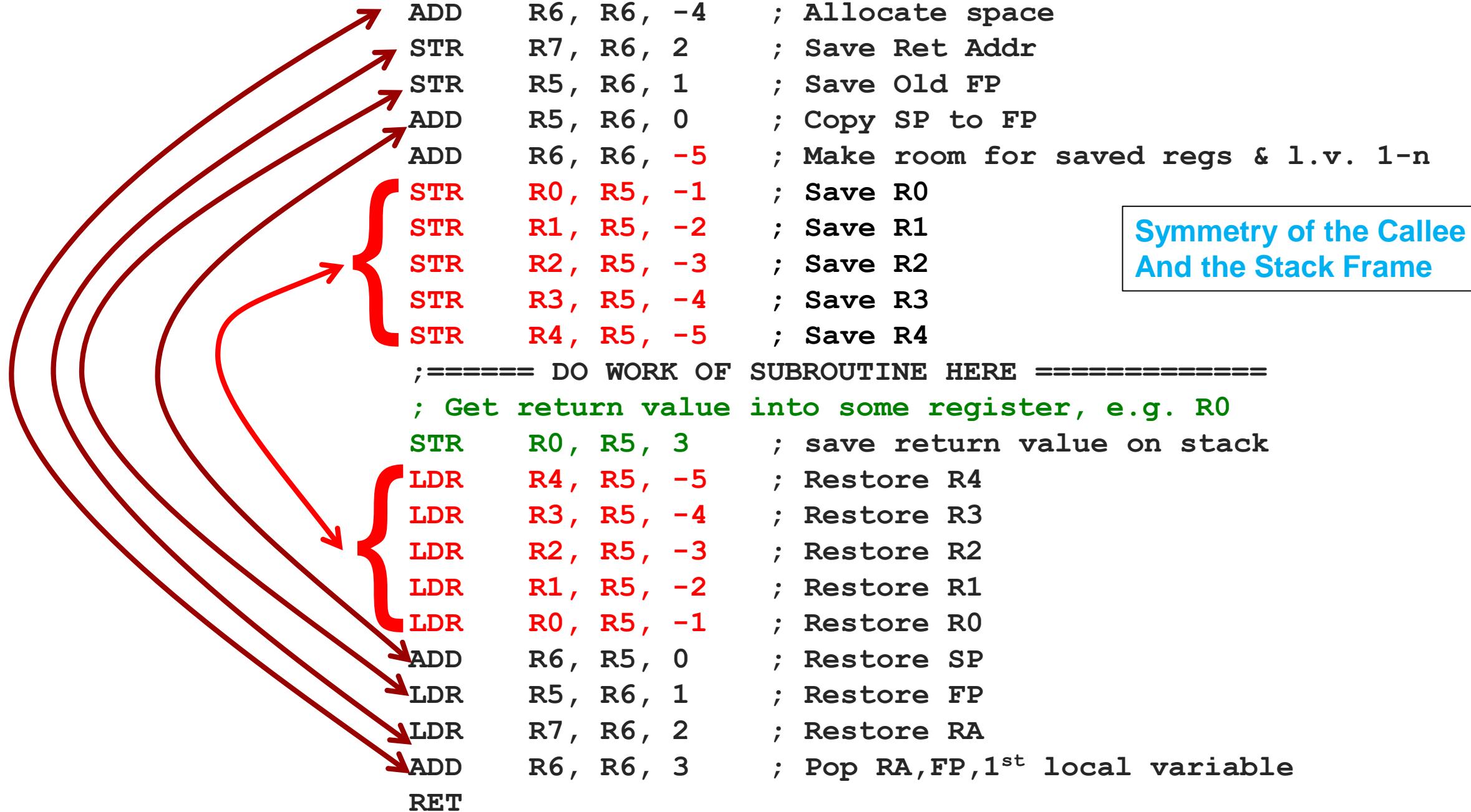
Stack Frame Progression



LC-3 Calling Convention

```
y = foo(a, b, c);
```

- ↗ Caller: Push args onto stack **right to left!**
- ↗ Caller: Jump to subroutine
- ↗ Callee: Decrement SP to leave four slots (ret value, ret addr, old FP, local var)
- ↗ Callee: Save copy of R7 (Ret Addr), and copy of R5 (Old FP)
- ↗ Callee: Save Callee: Set R5 (frame pointer) to be R6 (stack pointer)
- ↗ Callee: Allocate space (SP) for local variables & saved registers (R0-R4)
- ↗ Callee: Save registers R0-R4 used by the function
- ↗ Callee: Execute the code in the function
- ↗ Callee: Save the Ret Val (at R5 + 3)
- ↗ Callee: Restore saved registers (R0-R4)
- ↗ Callee: Set SP to FP, to pop off local vars and saved registers
- ↗ Callee: Restore the Ret Addr (to R7), and old Frame Pointer (to R5)
- ↗ Callee: Pop off 3 words (ret addr, old FP, first local var)
- ↗ Caller: Grab the Ret(urn) Val(ue)
- ↗ Caller: Deallocate space for Ret Val and args



Three Oddities

- ↗ We always allocate space for no less than one local variable, even when we need none
- ↗ The caller always pushes N words of arguments, but always pops $N+1$ words (which includes the Return Value)
- ↗ Which means the callee pushes M words of stack frame, but pops $M-1$ to leave the Return Value on top of the stack

How might the stack frame change?

Is the stack frame the same for every subroutine?

- ↗ If you only have one local variable, your stack frame will always look like the prior examples.
- ↗ If you have two or more local variables, you'll need to allocate more space and move the saved register locations up – to make room for the additional local vars.

Saving registers

- You always save R7 and R5
 - The old Return Address, and old Frame Pointer
- Easy answer: always save R0-R4 on the stack frame
 - The five general purpose registers.
- Optional:
 - If your subroutine only needs to use two registers (e.g. R1,R2), then you don't have to save the other regs (R0, R3, R4)
 - But you have to know which registers you'll use first
 - It is easier just to always save all five (R0-R4) in your stack frame

0000 The stack frame when the callee starts doing work:

SP(R6)->	Saved Reg #N
	...
	Saved Reg #1
	Local variable L
	...
FP(R5)->	Local variable 1
	<i>Old FP</i>
	<i>Return address (RA)</i>
	Return value (RV)
	arg 1
	...
	arg N

FFFF

Item	Location
Lv L	FP-L-1
RA	FP+2
RV	FP+3
Arg N	FP+N+3

Questions?

There's Code in the Next Slides

- ☞ The upcoming slides have example code in them to implement the LC-3 calling convention for a subroutine.
- ☞ Not only should you feel free to copy the code for your own programs, I strongly recommend that you copy this code.
- ☞ Reinventing the wheel can cost you many hours. Avoid it.
- ☞ Pay attention to the places you need to customize based on number of local variables, number of arguments, (and number of saved registers).

Example: mult(a,b)

- ↗ Presume you have a subroutine call
 - ↗ $m = \text{mult}(x, 3)$
 - ↗ *(assume m and x are labels in the caller)*

- ↗ And elsewhere a subroutine definition

```
int mult(int a, int b) {  
    int answer = 0;  
    while (a>0) {  
        answer += b;  
        a--;  
    }  
    return answer;  
}
```

- ↗ So how do we implement the subroutine, and the caller?

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
        ; mult(x, 3)
```

Caller

Pointers	Value	Purpose	Base+Offset
SP->			

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
        ; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
        ADD    R0, R0, 3
```

Pointers	Value	Purpose	Base+Offset
SP->			

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
ADD      R0, R0, 3  
ADD      R6, R6, -1
```

Pointers	Value	Purpose	Base+Offset
SP->			

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
ADD      R0, R0, 3  
ADD      R6, R6, -1  
STR      R0, R6, 0
```

Pointers	Value	Purpose	Base+Offset
SP->	3	arg 2 (b)	

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
ADD      R0, R0, 3  
ADD      R6, R6, -1  
STR      R0, R6, 0  
LD       R0, X      ; push(x)
```

Pointers	Value	Purpose	Base+Offset
SP->	3	arg 2 (b)	

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
ADD     R0, R0, 3  
ADD     R6, R6, -1  
STR    R0, R6, 0  
LD     R0, X      ; push(x)  
ADD     R6, R6, -1
```

Pointers	Value	Purpose	Base+Offset
SP->			
	3	arg 2 (b)	

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
          ADD    R0, R0, 3  
          ADD    R6, R6, -1  
          STR   R0, R6, 0  
          LD     R0, X      ; push(x)  
          ADD   R6, R6, -1  
          STR   R0, R6, 0
```

Pointers	Value	Purpose	Base+Offset
SP->	x	arg 1 (a)	
	3	arg 2 (b)	

Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
Caller ; push arguments in reverse order  
; mult(x, 3)
```

```
Caller AND    R0, R0, 0 ; push(3)  
ADD     R0, R0, 3  
ADD     R6, R6, -1  
STR    R0, R6, 0  
LD     R0, X      ; push(x)  
ADD     R6, R6, -1  
STR    R0, R6, 0  
JSR    MULT       ; mult(x, 3)
```

Pointers	Value	Purpose	Base+Offset
SP->	x	arg 1 (a)	
	3	arg 2 (b)	

And off we go to the address of mult()

Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Callee - the subroutine itself  
  
; First, lay out our stack frame!
```

Pointers	Value	Purpose	Base+Offset
SP->	x	arg 1 (a)	
	3	arg 2 (b)	

Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Callee - the subroutine itself  
  
; First, lay out our stack frame!  
  
; This is always the same for 2  
; args, 1 local variable, and  
; 5 saved registers  
  
; For other counts, just adjust the  
; amount of stack space as needed
```

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	
		Saved Reg 4	
		Saved Reg 3	
		Saved Reg 2	
		Saved Reg 1	
		Local Var (answer)	
		Old Frame Ptr	
		Old Ret Address	
		Return Value	
SP->	x	arg 1 (a)	
	3	arg 2 (b)	

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	
		Saved Reg 4	
		Saved Reg 3	
		Saved Reg 2	
		Saved Reg 1	
		Local Var (answer)	
		Old Frame Ptr	
		Old Ret Address	
		Return Value	
SP->	x	arg 1 (a)	
	3	arg 2 (b)	

0000



FFFF



Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
MULT    ADD      R6, R6, -4 ; push 4 wds  
          ; set rv later
```

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	
		Saved Reg 4	
		Saved Reg 3	
		Saved Reg 2	
		Saved Reg 1	
SP->	???	Local Var (answer)	
	???	Old Frame Ptr	
	???	Old Ret Address	
	???	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
MULT    ADD      R6, R6, -4 ; push 4 wds  
          ; set rv later  
STR     R7, R6, 2  ; save RA
```

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	
		Saved Reg 4	
		Saved Reg 3	
		Saved Reg 2	
		Saved Reg 1	
SP->	???	Local Var (answer)	
	???	Old Frame Ptr	
	old R7	Old Ret Address	
	???	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: **2** arguments, **1** local variable, **5** saved registers)

```
MULT    ADD     R6, R6, -4 ; push 4 wds  
          ; set rv later  
STR     R7, R6, 2  ; save RA  
STR     R5, R6, 1  ; save old FP  
          ; set local var later
```

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	
		Saved Reg 4	
		Saved Reg 3	
		Saved Reg 2	
		Saved Reg 1	
SP->	???	Local Var (answer)	
	old R5	Old Frame Ptr	
	old R7	Old Ret Address	
	???	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: **2** arguments, **1** local variable, **5** saved registers)

```

MULT   ADD    R6, R6, -4 ; push 4 wds
              ; set rv later
STR     R7, R6, 2  ; save RA
STR     R5, R6, 1  ; save old FP
              ; set local var later
ADD     R5, R6, 0  ; FP = SP

```

	FP->		0000 ↑	
	Pointers	Value	Purpose	Base+Offset
			Saved Reg 5	
			Saved Reg 4	
			Saved Reg 3	
			Saved Reg 2	
			Saved Reg 1	
	SP->	???	Local Var (answer)	
		old R5	Old Frame Ptr	
		old R7	Old Ret Address	
		???	Return Value	
		x	arg 1 (a)	
		3	arg 2 (b)	
			↓ FFFF	

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT	ADD	R6, R6, -4 ; push 4 wds ; set rv later
	STR	R7, R6, 2 ; save RA
	STR	R5, R6, 1 ; save old FP ; set local var later
	ADD	R5, R6, 0 ; FP = SP

Pointers	Value	Purpose	Base+Offset
		Saved Reg 5	R5, -5
		Saved Reg 4	R5, -4
		Saved Reg 3	R5, -3
		Saved Reg 2	R5, -2
		Saved Reg 1	R5, -1
FP->	SP->	??? Local Var (answer)	R5, 0
		old R5 Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	??? Return Value		R5, 3
	x arg 1 (a)		R5, 4
	3 arg 2 (b)		R5, 5
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT	ADD	R6, R6, -4 ; push 4 wds ; set rv later
	STR	R7, R6, 2 ; save RA
	STR	R5, R6, 1 ; save old FP ; set local var later
	ADD	R5, R6, 0 ; FP = SP
	ADD	R6, R6, -5 ; push 5 words

Pointers	Value	Purpose	Base+Offset
SP->		Saved Reg 5	R5, -5
		Saved Reg 4	R5, -4
		Saved Reg 3	R5, -3
		Saved Reg 2	R5, -2
		Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT	ADD	R6, R6, -4 ; push 4 wds ; set rv later
	STR	R7, R6, 2 ; save RA
	STR	R5, R6, 1 ; save old FP ; set local var later
	ADD	R5, R6, 0 ; FP = SP
	ADD	R6, R6, -5 ; push 5 words
	STR	R0, R5, -1 ; save SR1

Pointers	Value	Purpose	Base+Offset
SP->		Saved Reg 5	R5, -5
		Saved Reg 4	R5, -4
		Saved Reg 3	R5, -3
		Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT		R6, R6, -4 ; push 4 wds	
		;	set rv later
STR		R7, R6, 2 ; save RA	
STR		R5, R6, 1 ; save old FP	
		;	set local var later
ADD		R5, R6, 0 ; FP = SP	
ADD		R6, R6, -5 ; push 5 words	
STR		R0, R5, -1 ; save SR1	
STR		R1, R5, -2 ; save SR2	

Pointers	Value	Purpose	Base+Offset
SP->		Saved Reg 5	R5, -5
		Saved Reg 4	R5, -4
		Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



↓

FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT		R6, R6, -4 ; push 4 wds	
		;	set rv later
STR		R7, R6, 2 ; save RA	
STR		R5, R6, 1 ; save old FP	
		;	set local var later
ADD		R5, R6, 0 ; FP = SP	
ADD		R6, R6, -5 ; push 5 words	
STR		R0, R5, -1 ; save SR1	
STR		R1, R5, -2 ; save SR2	
STR		R2, R5, -3 ; save SR3	

Pointers	Value	Purpose	Base+Offset
SP->		Saved Reg 5	R5, -5
		Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



↓

FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT	ADD	R6, R6, -4 ; push 4 wds
		; set rv later
STR		R7, R6, 2 ; save RA
STR		R5, R6, 1 ; save old FP
		; set local var later
ADD		R5, R6, 0 ; FP = SP
ADD		R6, R6, -5 ; push 5 words
STR		R0, R5, -1 ; save SR1
STR		R1, R5, -2 ; save SR2
STR		R2, R5, -3 ; save SR3
STR		R3, R5, -4 ; save SR4

Pointers	Value	Purpose	Base+Offset
SP->		Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Stack Buildup -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

MULT	ADD	R6, R6, -4 ; push 4 wds ; set rv later
	STR	R7, R6, 2 ; save RA
	STR	R5, R6, 1 ; save old FP ; set local var later
	ADD	R5, R6, 0 ; FP = SP
	ADD	R6, R6, -5 ; push 5 words
	STR	R0, R5, -1 ; save SR1
	STR	R1, R5, -2 ; save SR2
	STR	R2, R5, -3 ; save SR3
	STR	R3, R5, -4 ; save SR4
	STR	R4, R5, -5 ; save SR5

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

What next?

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;

    ;while (a>0) {
        ;answer += b;
        ;a--;
    }

    ;return answer;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0

    ;while (a>0) {
        ;answer += b;
        ;a--;
    }

    ;return answer;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	???	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
        ;answer += b;
        ;a--;
    }

    ;return answer;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	0	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1     LDR R0, R5, 4 ;R0 = a
        ;answer += b;
        ;a--;
    }
    ;return answer;
;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	0	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
    }

    ;return answer;
;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	0	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
    }
    BR W1
    ;return answer;
;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	0	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
        ;
    }
    BR W1
    ;return answer;
END_W1  NOP          ;placeholder
;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	0	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

After the loop, what value will be in answer (local variable)?

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
        ;
    }
    BR W1
    ;return answer;
END_W1  NOP          ;placeholder
;
}
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
        ;
        BR W1
        ;return answer;
END_W1 LDR R0, R5, 0 ;R0 = answer
;
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	???	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finally - we can do the work of mult()
;int mult(int a, int b) {
    ;int answer = 0;
    AND R0, R0, #0 ;R0 = 0
    STR R0, R5, 0 ;answer = R0
    ;while (a>0) {
W1        LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
            ;answer += b;
            ;a--;
        ;
    }
    BR W1
    ;return answer;
END_W1   LDR R0, R5, 0 ;R0 = answer
        STR R0, R5, 3 ;set ret val to answer
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;

        ;a--;
BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer

        ;a--;
BR_W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ;a--;
        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ADD R0, R0, R1 ;R0 = answer+b

        ;a--;
        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



↓

FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ADD R0, R0, R1 ;R0 = answer+b
        STR R0, R5, 0 ;answer = R0
        ;a--;
        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ADD R0, R0, R1 ;R0 = answer+b
        STR R0, R5, 0 ;answer = R0
        ;a--;
        LDR R0, R5, 4 ;R0 = a

        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ADD R0, R0, R1 ;R0 = answer+b
        STR R0, R5, 0 ;answer = R0
        ;a--;
        LDR R0, R5, 4 ;R0 = a
        ADD R0, R0, #-1 ;R0 = a-1
        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

0000



FFFF



Implementation -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Finish the body of the while loop

;while (a>0) {
W1    LDR R0, R5, 4 ;R0 = a
        BRnz END_W1
        ;answer += b;
        LDR R0, R5, 0 ;R0 = answer
        LDR R1, R5, 5 ;R1 = b
        ADD R0, R0, R1 ;R0 = answer+b
        STR R0, R5, 0 ;answer = R0
        ;a--;
        LDR R0, R5, 4 ;R0 = a
        ADD R0, R0, #-1 ;R0 = a-1
        STR R0, R5, 4 ;a = a-1
        BR W1
; }
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

What do we do next? (Are we ready to return?)

Stack Teardown -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; We need to tear down the stack frame

LDR R4, R5, -5 ; restore R4

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

Stack Teardown -- Callee: mult(int a, int b)

(This example: **2** arguments, **1** local variable, **5** saved registers)

; We need to tear down the stack frame

```
LDR    R4, R5, -5      ; restore R4
LDR    R3, R5, -4      ; restore R3
LDR    R2, R5, -3      ; restore R2
LDR    R1, R5, -2      ; restore R1
LDR    R0, R5, -1      ; restore R0
```

Pointers	Value	Purpose	Base+Offset
SP->	old R4	Saved Reg 5	R5, -5
	old R3	Saved Reg 4	R5, -4
	old R2	Saved Reg 3	R5, -3
	old R1	Saved Reg 2	R5, -2
	old R0	Saved Reg 1	R5, -1
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5
			FFFF

Stack Teardown -- Callee: mult(int a, int b)

(This example: **2** arguments, **1** local variable, **5** saved registers)

; We need to tear down the stack frame

```
LDR R4, R5, -5 ; restore R4  
LDR R3, R5, -4 ; restore R3  
LDR R2, R5, -3 ; restore R2  
LDR R1, R5, -2 ; restore R1  
LDR R0, R5, -1 ; restore R0  
ADD R6, R5, 0 ; pop saved regs,  
 ; and local vars    SP->
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

Stack Teardown -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; We need to tear down the stack frame

```
LDR R4, R5, -5 ; restore R4
LDR R3, R5, -4 ; restore R3
LDR R2, R5, -3 ; restore R2
LDR R1, R5, -2 ; restore R1
LDR R0, R5, -1 ; restore R0
ADD R6, R5, 0 ; pop saved regs,
; and local vars    SP->
LDR R7, R5, 2 ; R7 = ret addr
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
FP->	x*3	Local Var (answer)	R5, 0
	old R5	Old Frame Ptr	R5, 1
	old R7	Old Ret Address	R5, 2
	x*3	Return Value	R5, 3
	x	arg 1 (a)	R5, 4
	3	arg 2 (b)	R5, 5

Stack Teardown -- Callee: mult(int a, int b)

(This example: **2** arguments, **1** local variable, **5** saved registers)

; We need to tear down the stack frame

```
LDR    R4, R5, -5      ; restore R4  
LDR    R3, R5, -4      ; restore R3  
LDR    R2, R5, -3      ; restore R2  
LDR    R1, R5, -2      ; restore R1  
LDR    R0, R5, -1      ; restore R0  
ADD    R6, R5, 0        ; pop saved regs,  
                      ; and local vars  
LDR    R7, R5, 2        ; R7 = ret addr  
LDR    R5, R5, 1        ; FP = Old FP
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
SP->	x*3	Local Var (answer)	
	old R5	Old Frame Ptr	
	old R7	Old Ret Address	
	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

Stack Teardown -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; We need to tear down the stack frame

```
LDR    R4, R5, -5      ; restore R4
LDR    R3, R5, -4      ; restore R3
LDR    R2, R5, -3      ; restore R2
LDR    R1, R5, -2      ; restore R1
LDR    R0, R5, -1      ; restore R0
ADD    R6, R5, 0        ; pop saved regs,
                      ; and local vars
LDR    R7, R5, 2        ; R7 = ret addr
LDR    R5, R5, 1        ; FP = Old FP
ADD    R6, R6, 3        ; pop 3 words
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
SP->	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

And now we're finally ready to return!

Stack Teardown -- Callee: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; We need to tear down the stack frame

```
LDR R4, R5, -5 ; restore R4  
LDR R3, R5, -4 ; restore R3  
LDR R2, R5, -3 ; restore R2  
LDR R1, R5, -2 ; restore R1  
LDR R0, R5, -1 ; restore R0  
ADD R6, R5, 0 ; pop saved regs,  
 ; and local vars  
LDR R7, R5, 2 ; R7 = ret addr  
LDR R5, R5, 1 ; FP = Old FP  
ADD R6, R6, 3 ; pop 3 words  
RET ; mult() is done!
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
SP->	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

Where do we go now?

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```
; m = mult(x, 3);  
  
; push(3)  
; push(x)  
JSR MULT      ;call mult(x,3)
```

From the caller's perspective:

JSR appeared to push the return value onto the stack!

What's changed since we left?

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
SP->	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Back to the caller, after JSR

; m = mult(x, 3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
SP->	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

```
; Back to the caller, after JSR

; m = mult(x, 3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
LDR R0, R6, 0 ;R0 = return value
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
SP->	x*3	Return Value	
	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```
; m = mult(x, 3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)
```

```
;pop the return value off the stack
LDR R0, R6, 0    ;R0 = return value
ADD R6, R6, 1
```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
	x*3		
SP->	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```

; m = mult(x,3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
LDR R0, R6, 0  ;R0 = return value
ADD R6, R6, 1
;save return value at label m

```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
	x*3		
SP->	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```

; m = mult(x, 3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
LDR R0, R6, 0  ;R0 = return value
ADD R6, R6, 1
;save return value at label m
ST R0, m

```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
	x*3		
SP->	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```

; m = mult(x,3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
LDR R0, R6, 0  ;R0 = return value
ADD R6, R6, 1
;save return value at label m
ST R0, m
;pop the arguments off the stack

```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
	x*3		
SP->	x	arg 1 (a)	
	3	arg 2 (b)	
			FFFF

After JSR – Caller: mult(int a, int b)

(This example: 2 arguments, 1 local variable, 5 saved registers)

; Back to the caller, after JSR

```

; m = mult(x,3);

; push(3)
; push(x)
JSR MULT      ;call mult(x,3)

;pop the return value off the stack
LDR R0, R6, 0    ;R0 = return value
ADD R6, R6, 1
;save return value at label m
ST R0, m
;pop the arguments off the stack
ADD R6, R6, 2

```

Pointers	Value	Purpose	Base+Offset
	old R4		
	old R3		
	old R2		
	old R1		
	old R0		
	x*3		
	old R5		
	old R7		
	x*3		
	x		
	3		
SP->			FFFF

Now the stack frame is right back where we started!

Question

In what order does a subroutine push items to create its stack frame?

- A. Return address, Return value, Saved FP, local variables, saved R0-R4
- B. Return value, Return address, Saved FP, local variables, saved R0-R4 
- C. Saved R0-R4, Local variables, Saved FP, Return address, Return value
- D. Return value, Return address, Saved FP, saved R0-R4, local variables

Today's number is 63,636

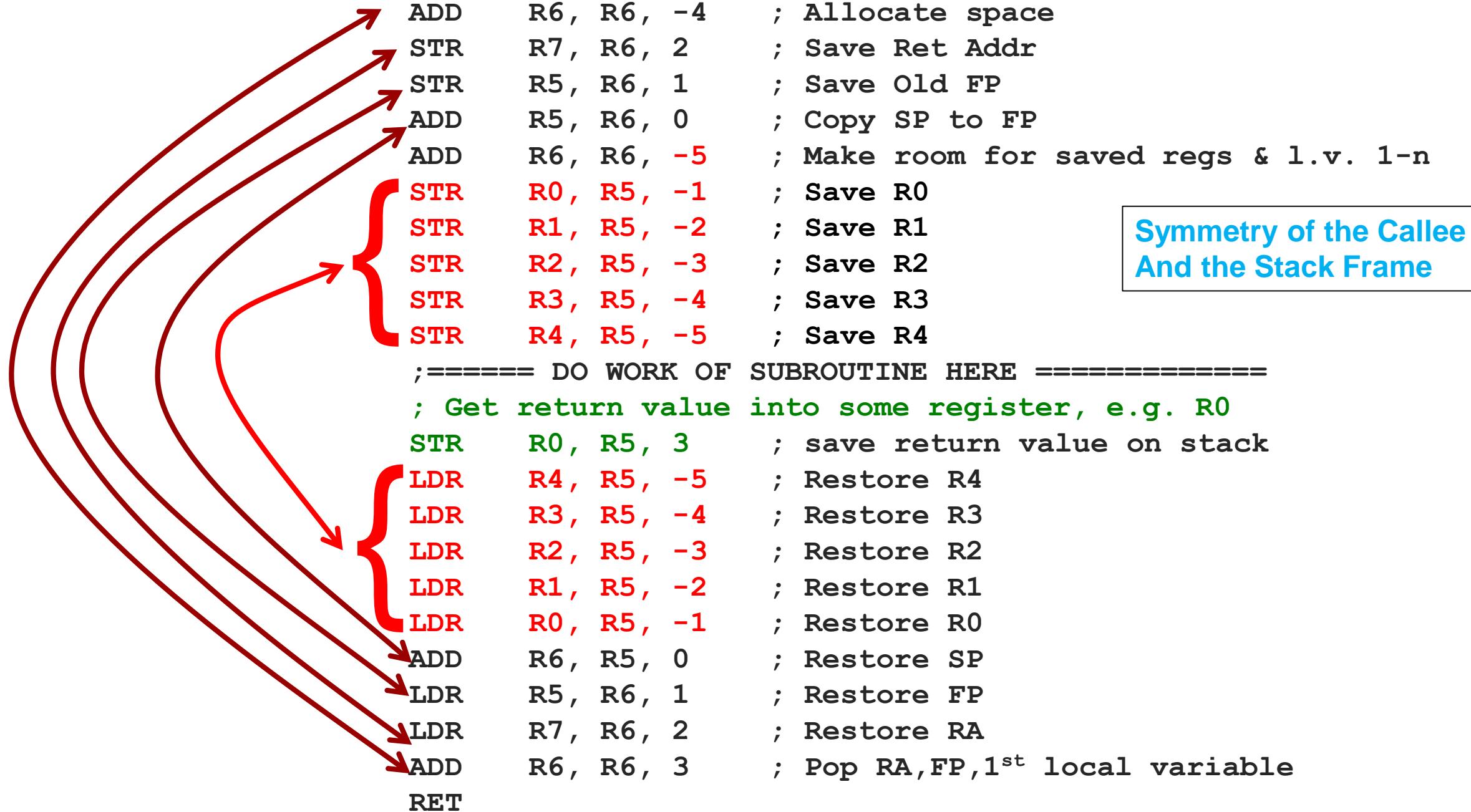
Review of Stack Frame, LC-3 Calling Convention



LC-3 Calling Convention

```
y = foo(a, b, c);
```

- ↗ Caller: Push args onto stack **right to left!**
- ↗ Caller: Jump to subroutine
- ↗ Callee: Decrement SP to leave four slots (ret value, ret addr, old FP, local var)
- ↗ Callee: Save copy of R7 (Ret Addr), and copy of R5 (Old FP)
- ↗ Callee: Save Callee: Set R5 (frame pointer) to be R6 (stack pointer)
- ↗ Callee: Allocate space (SP) for local variables & saved registers (R0-R4)
- ↗ Callee: Save registers R0-R4 used by the function
- ↗ Callee: Execute the code in the function
- ↗ Callee: Save the Ret Val (at R5 + 3)
- ↗ Callee: Restore saved registers (R0-R4)
- ↗ Callee: Set SP to FP, to pop off local vars and saved registers
- ↗ Callee: Restore the Ret Addr (to R7), and old Frame Pointer (to R5)
- ↗ Callee: Pop off 3 words (ret addr, old FP, first local var)
- ↗ Caller: Grab the Ret(urn) Val(ue)
- ↗ Caller: Deallocate space for Ret Val and args



Symmetry of the Callee
And the Stack Frame

Question

On the LC-3, where does R5 point in the stack frame during the execution of a subroutine?

- A. Return value
- B. Return address
- C. Saved FP
- D. First local variable 
- E. Last local variable
- F. First saved register

- ↗ Subroutines - overview
- ↗ The Stack
- ↗ LC-3 calling convention
 - ↗ Stack Pointer and Frame Pointer
 - ↗ Caller and Callee
 - ↗ Examples

There is Nothing Magical About Recursion

Or How Using a Stack Gives Us Recursion for Free



The function

```
int fact(int n)
{
    int answer;
    if (n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```

Question

When calculating a factorial using the algorithm described in this side deck, what is the maximum number of instances of FACT's stack frame on the stack when calculating fact(n)?

- A. 1
- B. 3
- C. $n + 1$ 
- D. n^2
- E. $n!$

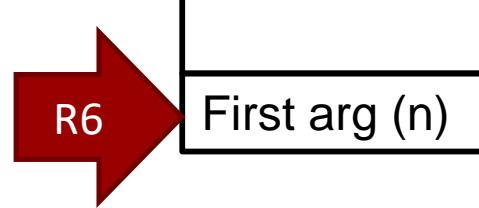
Assumptions

- ↗ Stack has been initialized and R6 is pointing to the top of the stack
 - ↗ What does this mean?
- ↗ The (assembly language) caller has placed arguments onto the stack right to left and then called the subroutine using a JSR or a JSRR thus the return address is already in R7.
- ↗ Our calling sequence will restore all of the registers (except R7) and the stack to the state they were in before the subroutine call, except for the return value.
- ↗ Thus the calling program can proceed just as if JSR/JSRR were really just a single instruction that pushes a return value onto the stack.

The initial code in a function is always the same

FACT

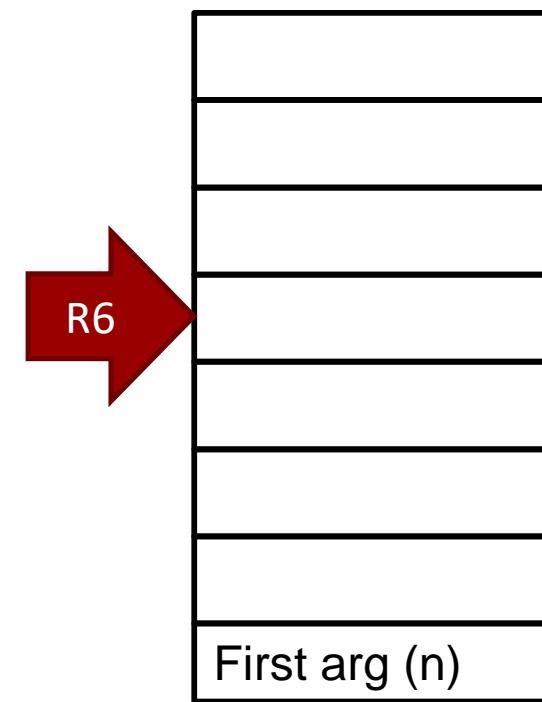
Because of our calling convention, we know that there will be an argument, n, on the stack when fact() is called and R7 will be set to the return address.



The initial part is always the same

FACT

ADD R6, R6, -4 ; Allocate space

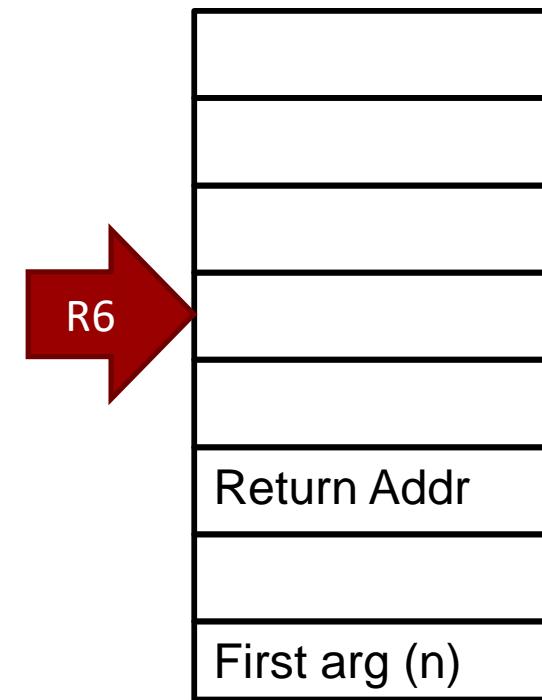


The initial part is always the same

FACT

ADD R6, R6, -4 ; Allocate space

STR R7, R6, 2 ; Save Ret Addr



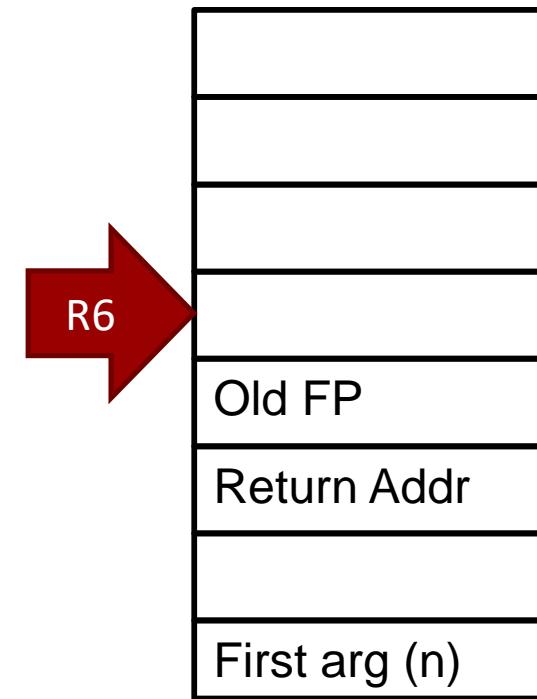
The initial part is always the same

FACT

ADD R6, R6, -4 ; Allocate space

STR R7, R6, 2 ; Save Ret Addr

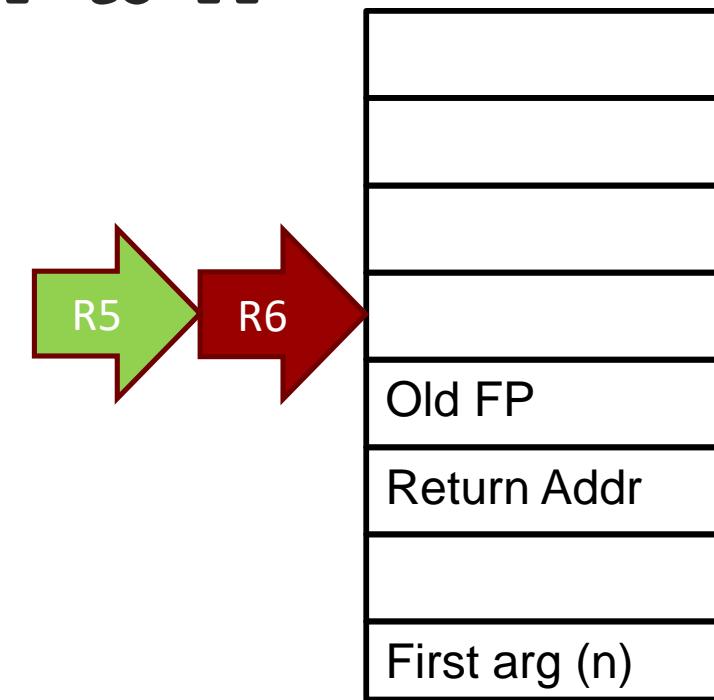
STR R5, R6, 1 ; Save Old FP



The initial part is always the same

FACT

```
ADD R6, R6, -4 ; Allocate space  
STR R7, R6, 2 ; Save Ret Addr  
STR R5, R6, 1 ; Save Old FP  
ADD R5, R6, 0 ; Copy SP to FP
```

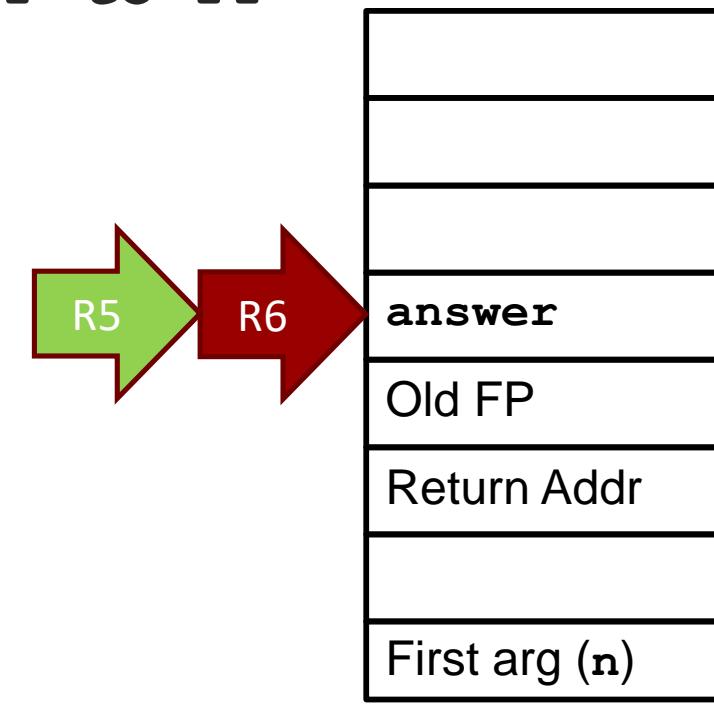


The initial code is always the same

FACT

```
ADD R6, R6, -4 ; Allocate space  
STR R7, R6, 2 ; Save Ret Addr  
STR R5, R6, 1 ; Save Old FP  
ADD R5, R6, 0 ; Copy SP to FP
```

Note: The space that is being pointed to by R5 and R6 can be used to store the first local variable, but it must be present. In our example we will put "answer" there.

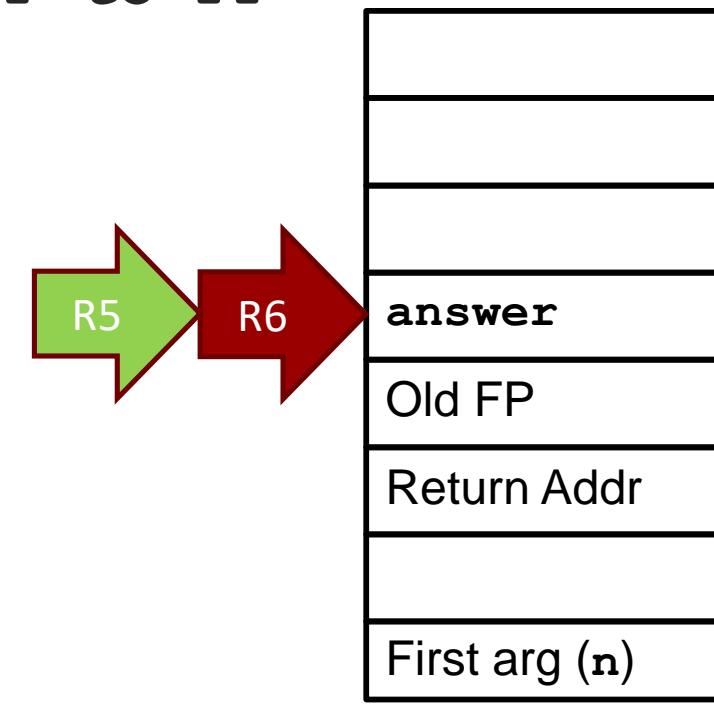


The initial code is always the same

FACT

```
ADD R6, R6, -4 ; Allocate space  
STR R7, R6, 2 ; Save Ret Addr  
STR R5, R6, 1 ; Save Old FP  
ADD R5, R6, 0 ; Copy SP to FP
```

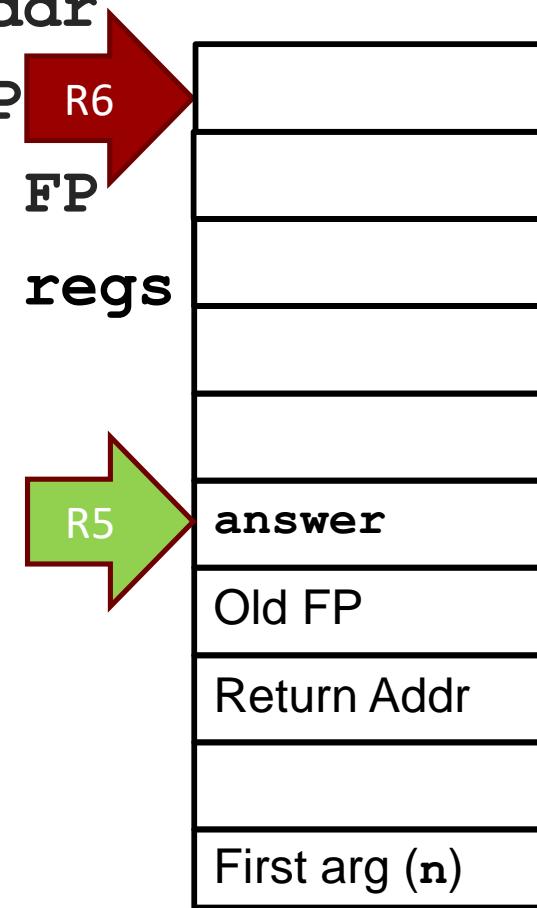
If there are more local variables we can move the stack pointer and make space for them here. In our example there are no more so we are good.



The initial code is always the same

FACT

```
ADD R6, R6, -4 ; Allocate space  
STR R7, R6, 2 ; Save Ret Addr  
STR R5, R6, 1 ; Save Old FP  
ADD R5, R6, 0 ; Copy SP to FP  
ADD R6, R6, -5; Room for 5 regs
```



The initial code is always the same

FACT

ADD R6, R6, -4 ; Allocate space

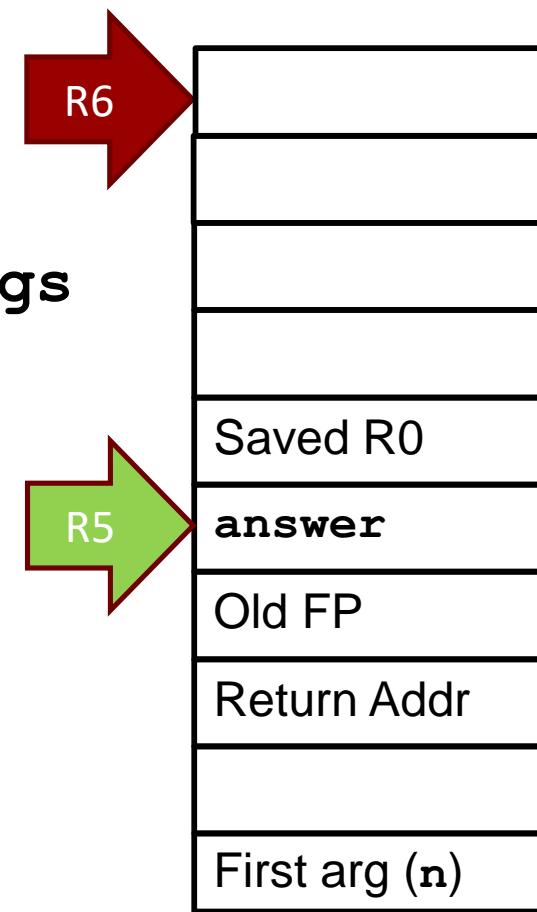
STR R7, R6, 2 ; Save Ret Addr

STR R5, R6, 1 ; Save Old FP

ADD R5, R6, 0 ; Copy SP to FP

ADD R6, R6, -5; Room for 5 regs

STR R0, R5, -1



The initial code is always the same

FACT

ADD R6, R6, -4 ; Allocate space

STR R7, R6, 2 ; Save Ret Addr

STR R5, R6, 1 ; Save Old FP

ADD R5, R6, 0 ; Copy SP to FP

ADD R6, R6, -5; Room for 5 regs

STR R0, R5, -1

STR R1, R5, -2

STR R2, R5, -3

STR R3, R5, -4

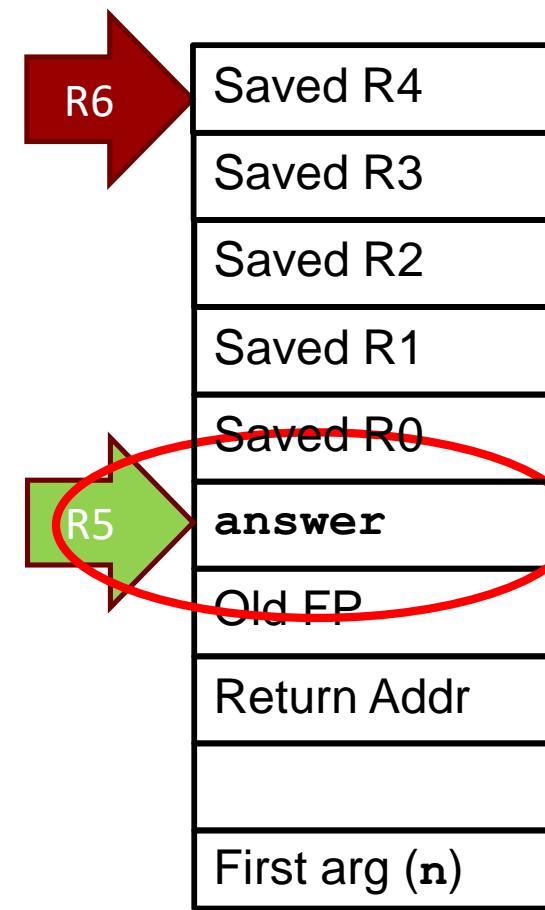
STR R4, R5, -5

R6	Saved R4
	Saved R3
	Saved R2
	Saved R1
	Saved R0
R5	answer
	Old FP
	Return Addr
	First arg (n)

The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer
```

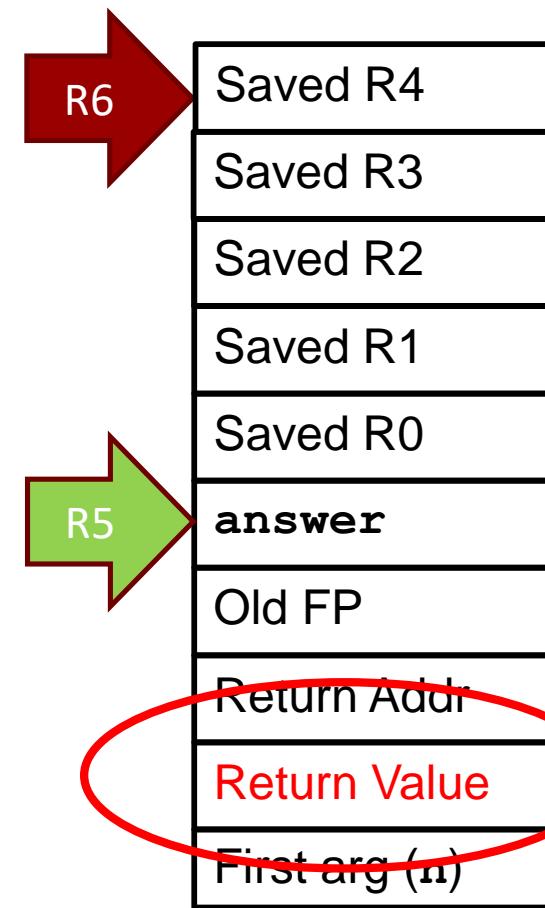
Assume our function has done all it is supposed to do including calculating and storing the final result in **answer**. Recall that in our example **answer** is just a variable.



The ending can be always the same

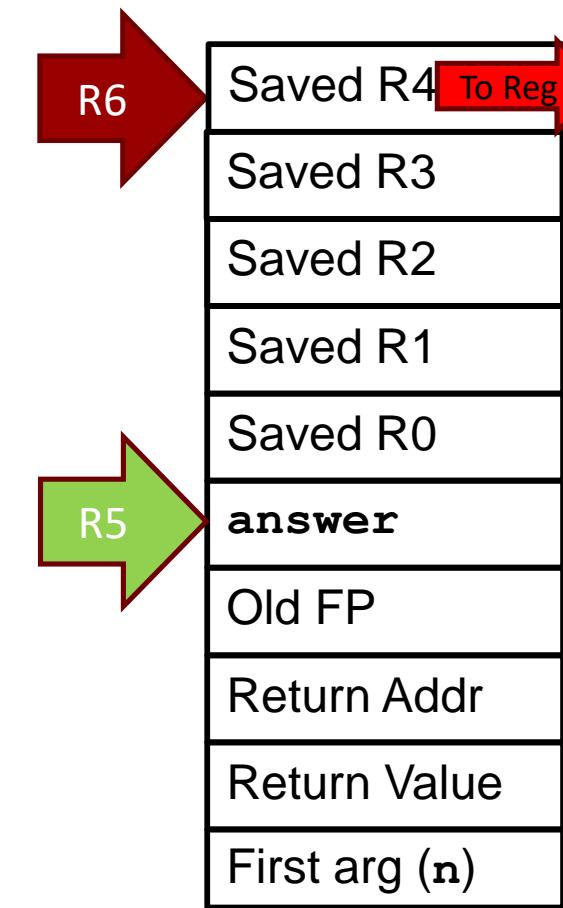
LDR R0, R5, 0 ; Ret val = answer

STR R0, R5, 3



The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer  
STR R0, R5, 3  
LDR R0, R5, -5; Restore R4
```



The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer
```

```
STR R0, R5, 3
```

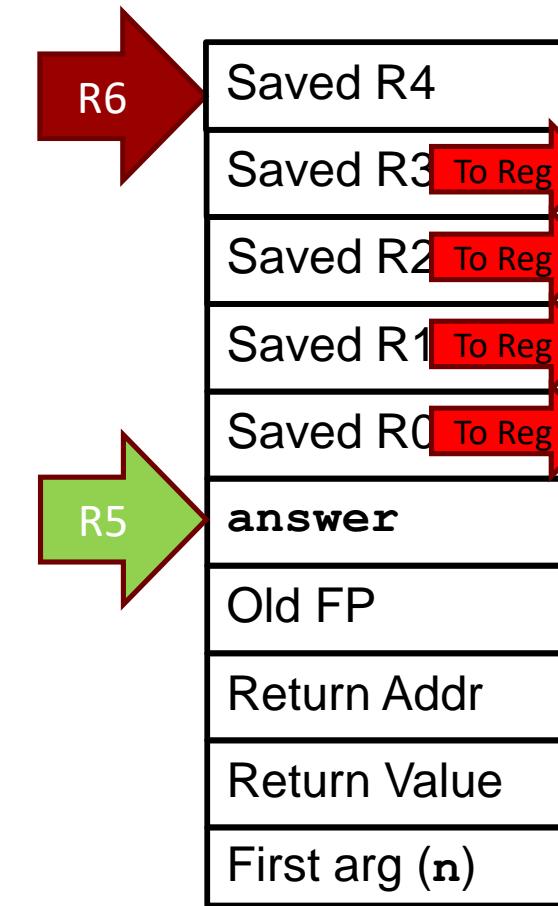
```
LDR R4, R5, -5; Restore R4
```

```
LDR R3, R5, -4; Restore R3
```

```
LDR R2, R5, -3; Restore R2
```

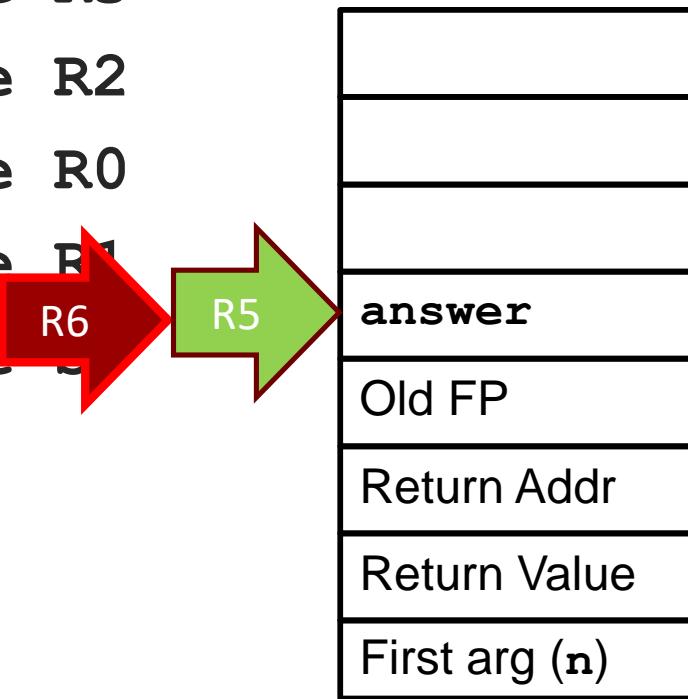
```
LDR R1, R5, -2; Restore R1
```

```
LDR R0, R5, -1; Restore R0
```



The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer  
STR R0, R5, 3  
LDR R4, R5, -5; Restore R4  
LDR R3, R5, -4; Restore R3  
LDR R2, R5, -3; Restore R2  
LDR R1, R5, -2; Restore R0  
LDR R0, R5, -1; Restore R1  
ADD R6, R5, 0 ; Restore
```



The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer
```

```
STR R0, R5, 3
```

```
LDR R4, R5, -5; Restore R4
```

```
LDR R3, R5, -4; Restore R3
```

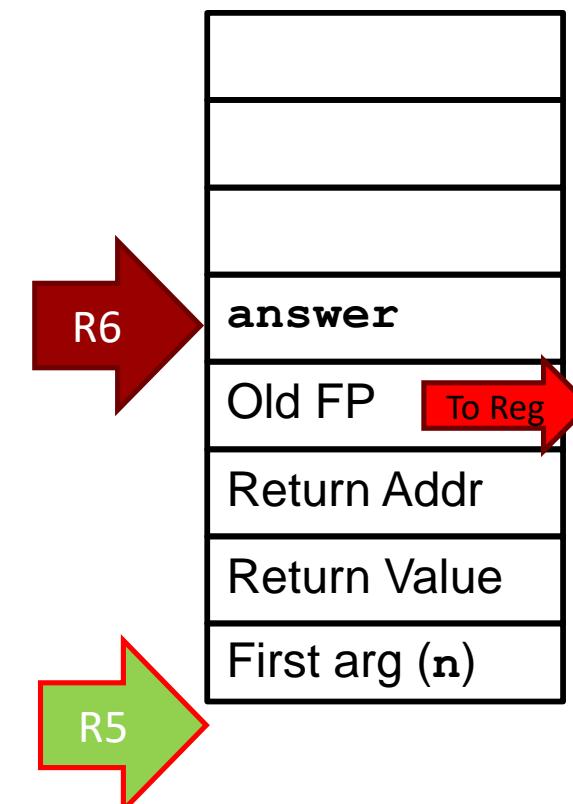
```
LDR R2, R5, -3; Restore R2
```

```
LDR R1, R5, -2; Restore R0
```

```
LDR R0, R5, -1; Restore R1
```

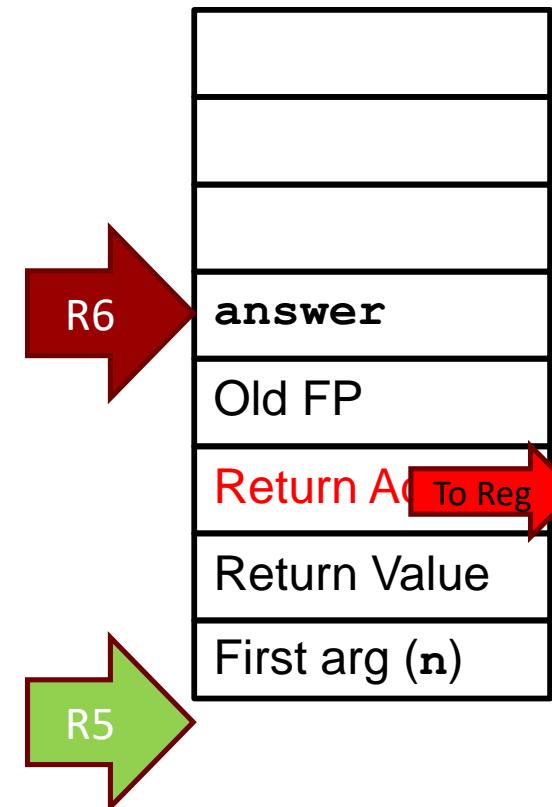
```
ADD R6, R5, 0 ; Restore SP
```

```
LDR R5, R6, 1 ; Restore FP
```



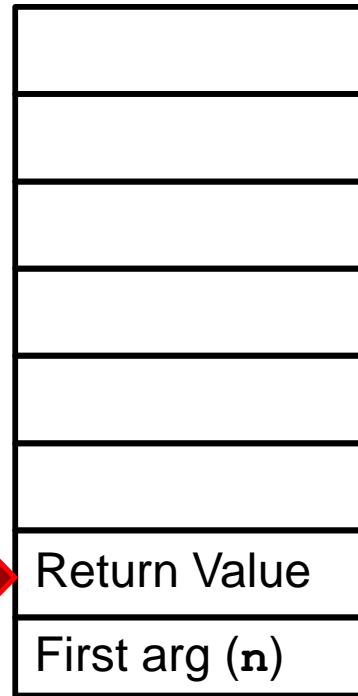
The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer  
STR R0, R5, 3  
LDR R4, R5, -5; Restore R4  
LDR R3, R5, -4; Restore R3  
LDR R2, R5, -3; Restore R2  
LDR R1, R5, -2; Restore R0  
LDR R0, R5, -1; Restore R1  
ADD R6, R5, 0 ; Restore SP  
LDR R5, R6, 1 ; Restore FP  
LDR R7, R6, 2 ; Restore RA
```



The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer  
STR R0, R5, 3  
LDR R4, R5, -5; Restore R4  
LDR R3, R5, -4; Restore R3  
LDR R2, R5, -3; Restore R2  
LDR R1, R5, -2; Restore R0  
LDR R0, R5, -1; Restore R1  
ADD R6, R5, 0 ; Restore SP  
LDR R5, R6, 1 ; Restore FP  
LDR R7, R6, 2 ; Restore RA  
ADD R6, R6, 3 ; Pop ra,fp,lv1
```



R5

R6

The ending can be always the same

```
LDR R0, R5, 0 ; Ret val = answer
```

```
STR R0, R5, 3
```

```
LDR R4, R5, -5; Restore R4
```

```
LDR R3, R5, -4; Restore R3
```

```
LDR R2, R5, -3; Restore R2
```

```
LDR R1, R5, -2; Restore R0
```

```
LDR R0, R5, -1; Restore R1
```

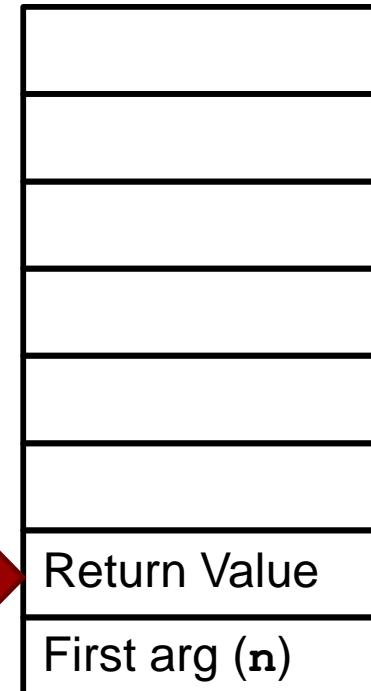
```
ADD R6, R5, 0 ; Restore SP
```

```
LDR R5, R6, 1 ; Restore FP
```

```
LDR R7, R6, 2 ; Restore RA
```

```
ADD R6, R6, 3 ; Pop ra,fp,lv1
```

```
RET
```



Everything we
just did took care
of the parts
marked in red

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```

FACT

```
ADD R6, R6, -4 ; Allocate space rv,ra,fp,lvl1
STR R7, R6, 2 ; Save Ret Addr
STR R5, R6, 1 ; Save Old FP
ADD R5, R6, 0 ; Copy SP to FP
ADD R6, R6, -5 ; Make room for saved regs & l.v. 1-n
STR R0, R5, -1 ; Save R0
STR R1, R5, -2 ; Save R1
STR R2, R5, -3 ; Save R2
STR R3, R5, -4 ; Save R3
STR R4, R5, -5 ; Save R4
;=====
LDR R0, R5, 0 ; Ret val = answer
STR R0, R5, 3 ;
LDR R4, R5, -5 ; Restore R4
LDR R3, R5, -4 ; Restore R3
LDR R2, R5, -3 ; Restore R2
LDR R1, R5, -2 ; Restore R1
LDR R0, R5, -1 ; Restore R0
ADD R6, R5, 0 ; Restore SP
LDR R5, R6, 1 ; Restore FP
LDR R7, R6, 2 ; Restore RA
ADD R6, R6, 3 ; Pop ra,fp,lvl1
RET
```

```
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```

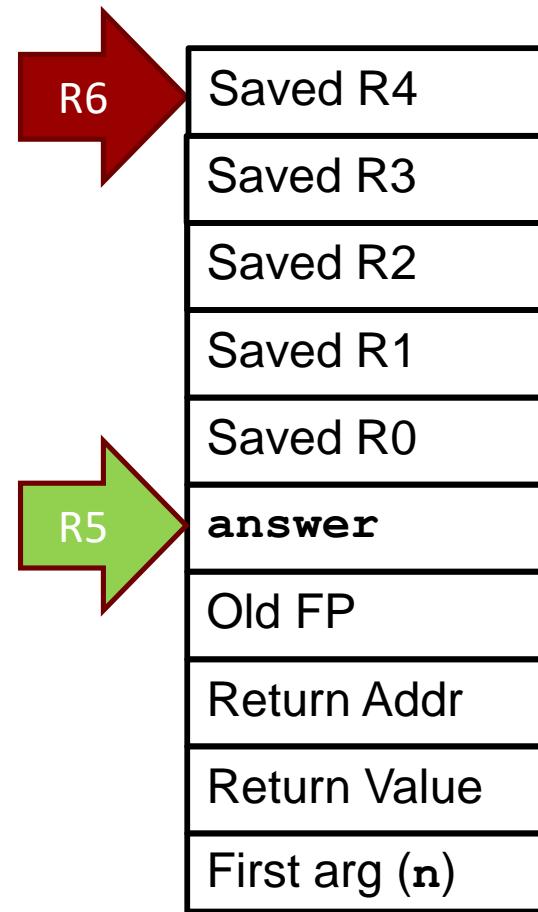
What about the function body?

When the code in *fact* starts, where is *n* being stored?

FP + 4

When we enter the body of *fact()*, the stack appears as it does at the right

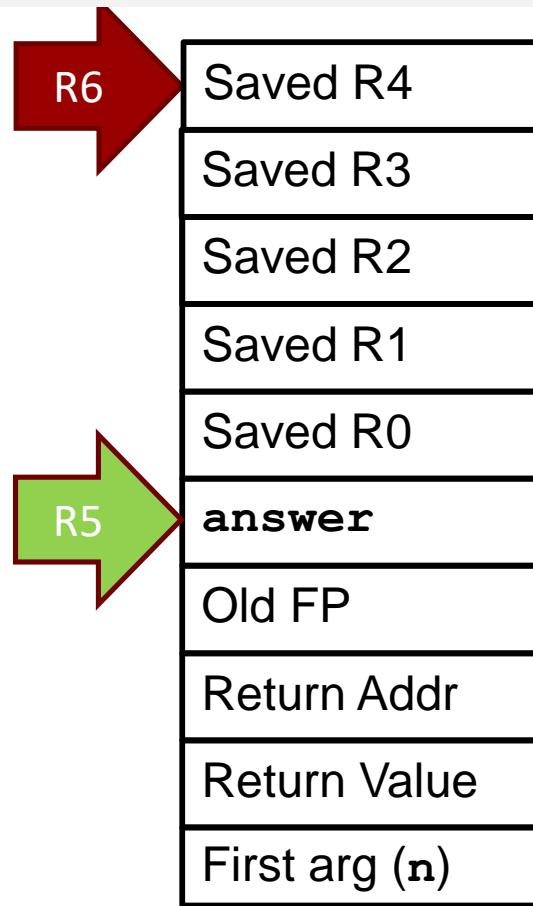
```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```



; Set up stack frame template goes above

;if (n <= 0) {

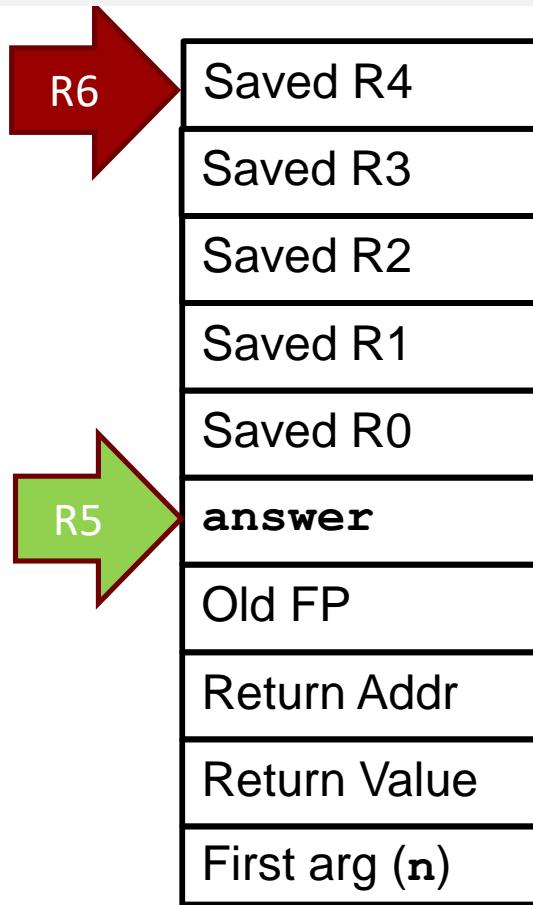
```
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}
```



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0
```

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```

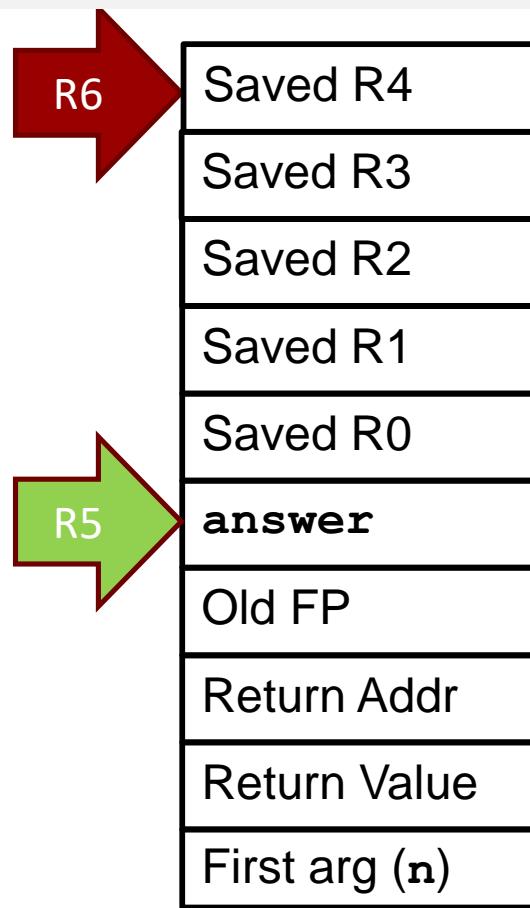


; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1
```

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```

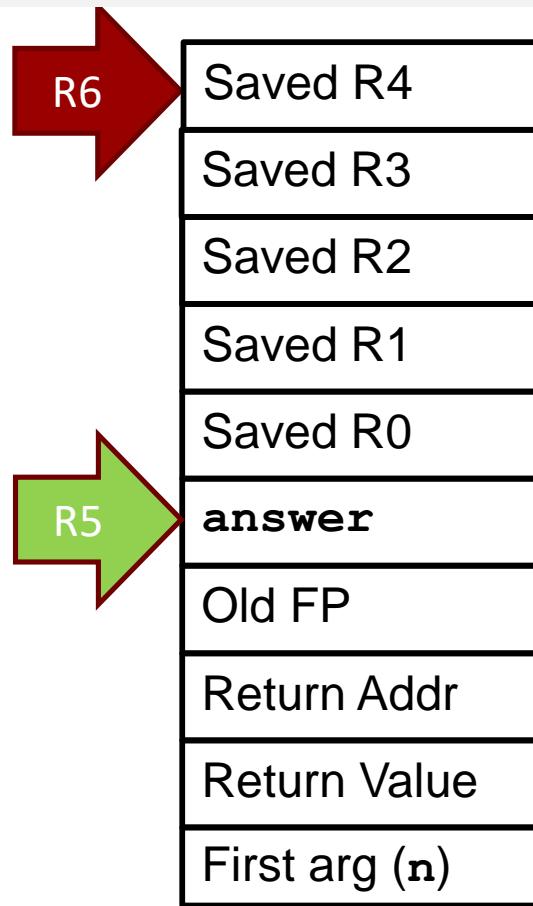
NOT IN CODE



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1  
  
;     answer = 1;
```

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```



```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```

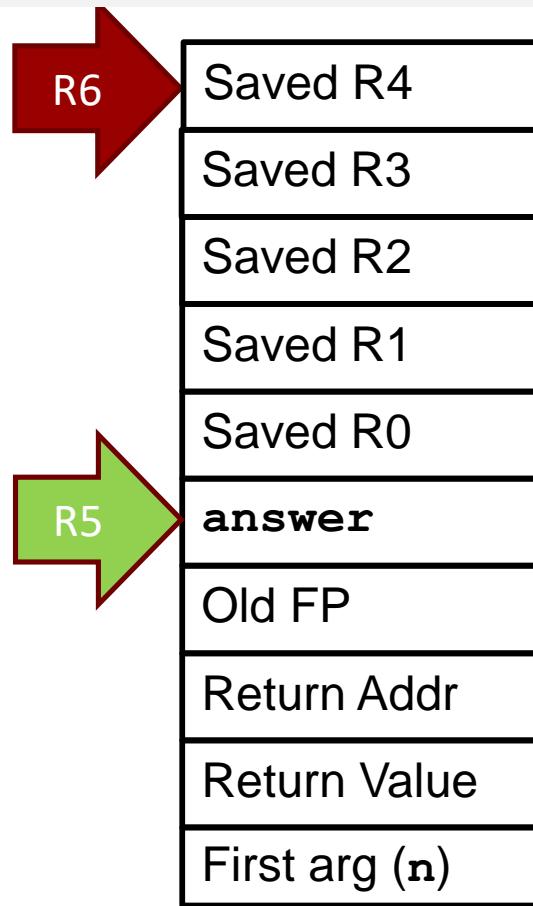
; Set up stack frame template goes above

```

;if (n <= 0) {
    LDR R0, R5, 4 ; n <= 0
    BRP IFELSE1

;     answer = 1;
    AND R0, R0, 0
    ADD R0, R0, 1

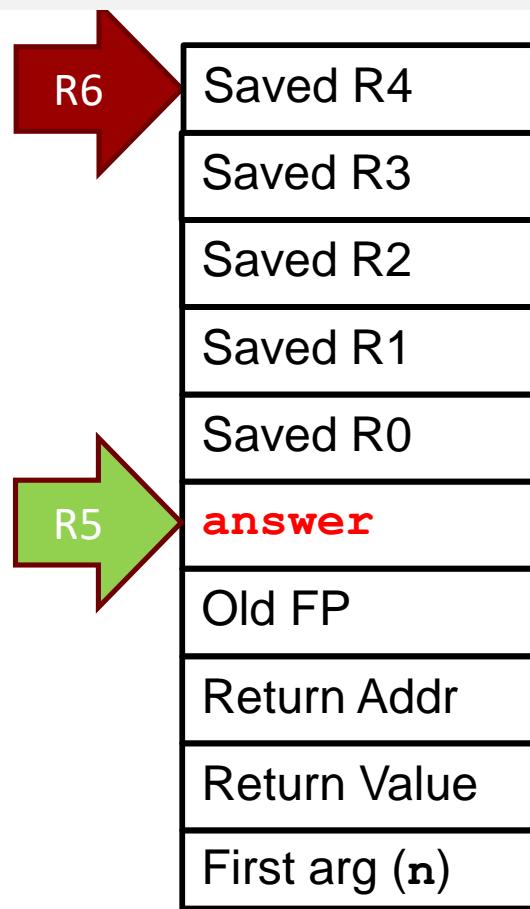
```



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1  
  
;    answer = 1;  
    AND R0, R0, 0  
    ADD R0, R0, 1  
    STR R0, R5, 0
```

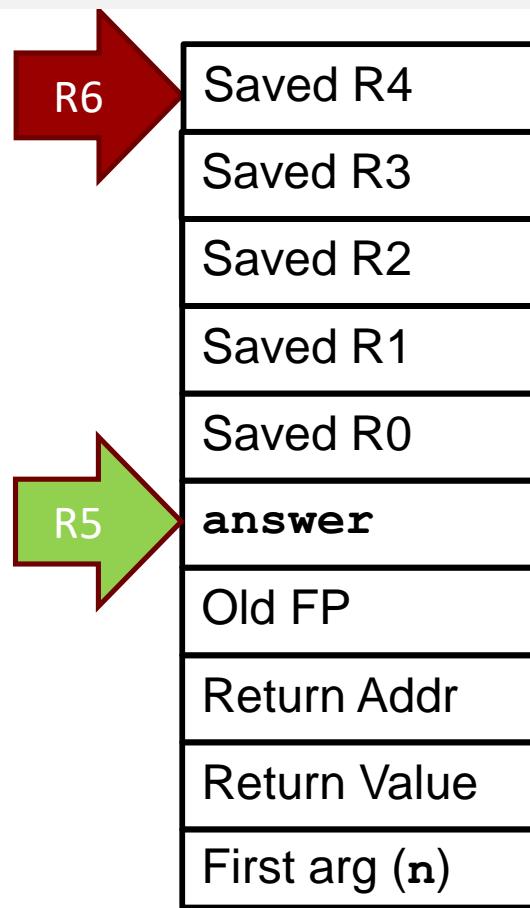
```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1  
  
;     answer = 1;  
    AND R0, R0, 0  
    ADD R0, R0, 1  
    STR R0, R5, 0  
;} else {
```

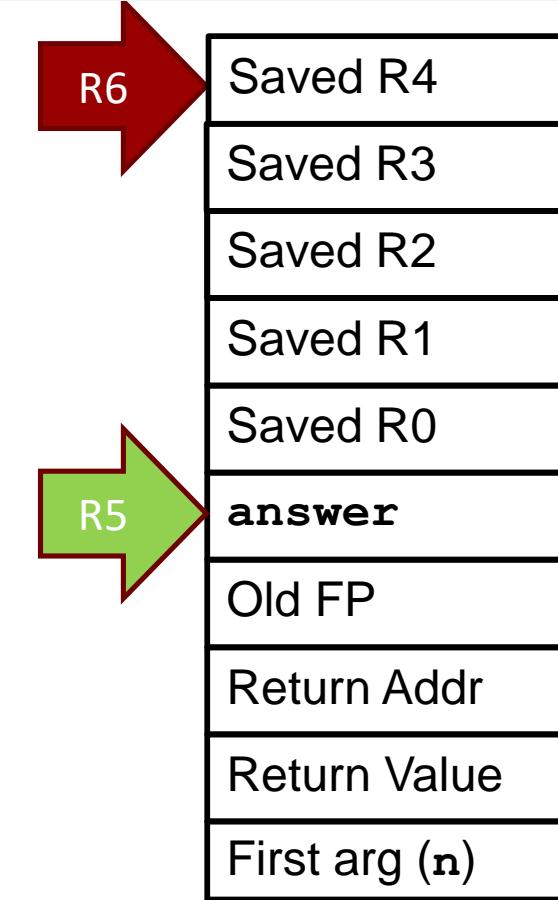
```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1  
  
;     answer = 1;  
    AND R0, R0, 0  
    ADD R0, R0, 1  
    STR R0, R5, 0  
  
;} else {  
    BR ENDIF1
```

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;
```



```

; Set up stack frame template goes above

;if (n <= 0) {
    LDR R0, R5, 4 ; n <= 0
    BRP IFELSE1

; answer = 1;
    AND R0, R0, 0
    ADD R0, R0, 1
    STR R0, R5, 0

;} else {
    BR ENDIF1

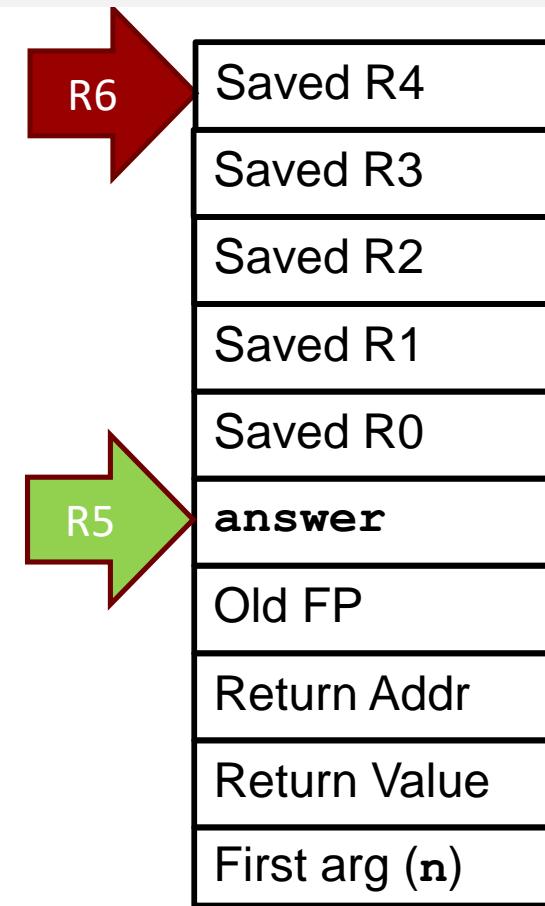
; answer = n * fact(n-1);

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

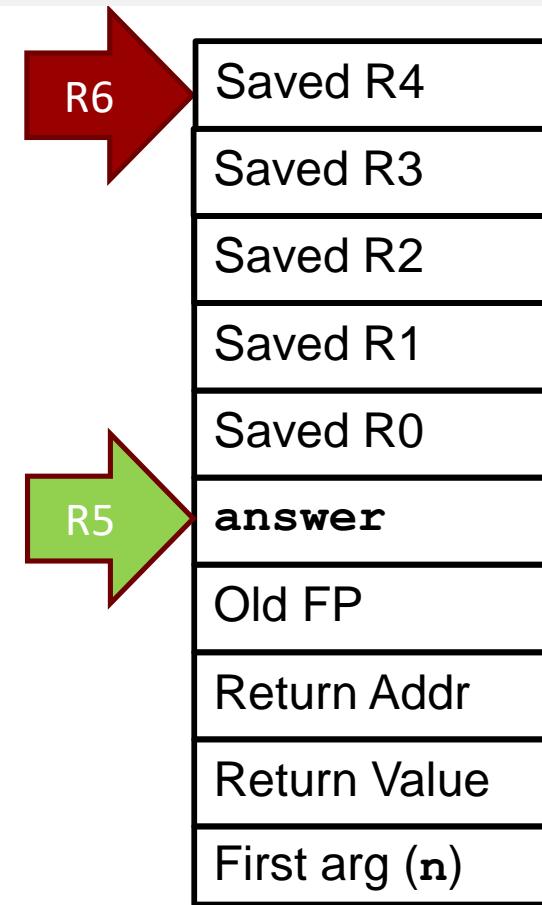
```



; Set up stack frame template goes above

```
;if (n <= 0) {  
    LDR R0, R5, 4 ; n <= 0  
    BRP IFELSE1  
  
    answer = 1;  
    AND R0, R0, 0  
    ADD R0, R0, 1  
    STR R0, R5, 0  
  
} else {  
    BR ENDIF1  
  
    answer = n * fact(n-1);  
  
; rewrite: R0 = fact(n-1); answer = mult(n, R0);
```

```
int fact(int n) {  
    int answer;  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = n * fact(n-1);  
    return answer;  
}
```



; Set up stack frame template goes above

```

;if (n <= 0) {
    LDR R0, R5, 4 ; n <= 0
    BRP IFELSE1

;     answer = 1;
    AND R0, R0, 0
    ADD R0, R0, 1
    STR R0, R5, 0

;} else {
    BR ENDIF1

;     answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0);

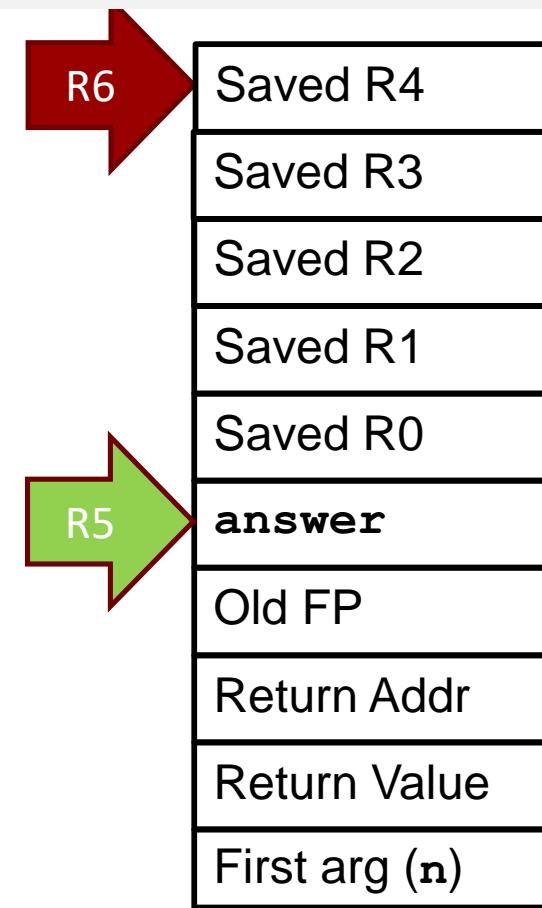
IFELSE1    LDR R0, R5, 4 ; Push n-1

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

; Set up stack frame template goes above

;if (n <= 0) {
    LDR R0, R5, 4 ; n <= 0
    BRP IFELSE1

; answer = 1;
    AND R0, R0, 0
    ADD R0, R0, 1
    STR R0, R5, 0

; } else {
    BR ENDIF1

; answer = n * fact(n-1);

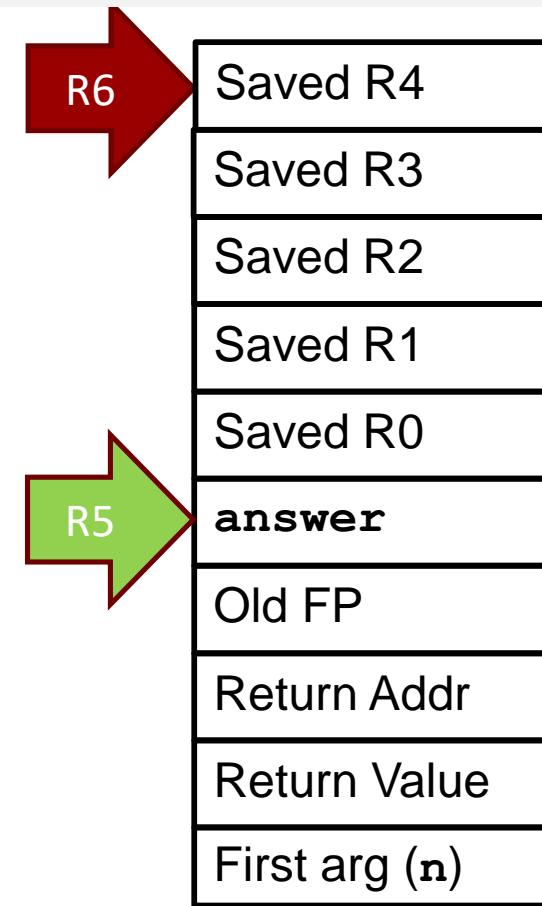
; rewrite: R0 = fact(n-1); answer = mult(n, R0);
IFELSE1   LDR R0, R5, 4 ; Push n-1
    ADD R0, R0, -1

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

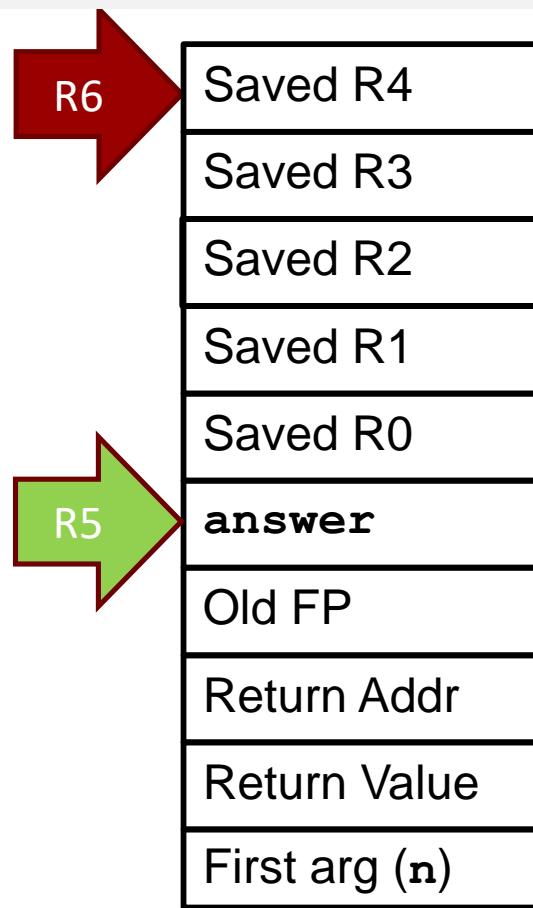
;     answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1    LDR    R0, R5, 4 ; Push n-1
            ADD    R0, R0, -1
            ADD    R6, R6, -1

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

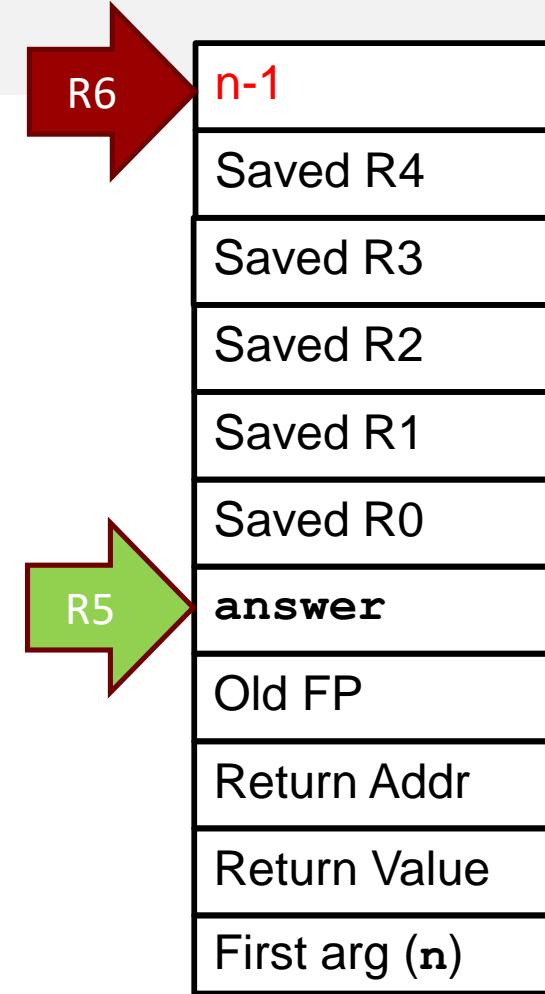
;     answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR   R0, R6, 0

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

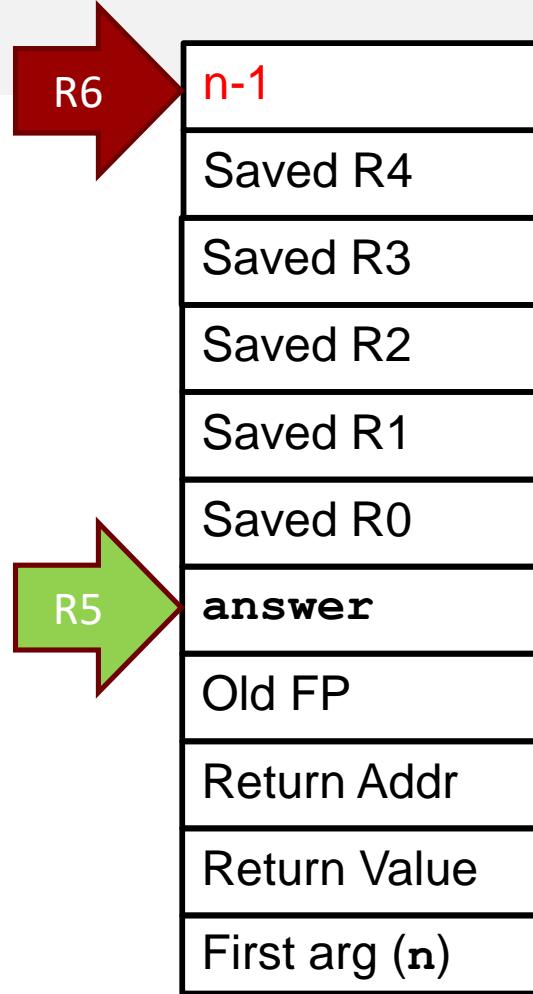
;      answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1    LDR    R0, R5, 4 ; Push n-1
            ADD    R0, R0, -1
            ADD    R6, R6, -1
            STR    R0, R6, 0

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



Now, how do you call a subroutine?

```

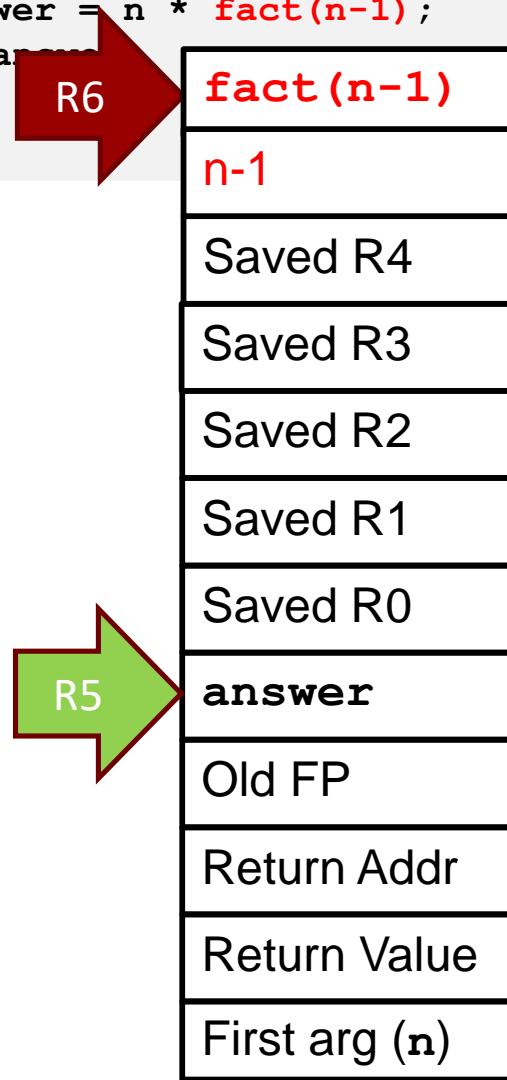
;      answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1    LDR    R0, R5, 4 ; Push n-1
            ADD    R0, R0, -1
            ADD    R6, R6, -1
            STR    R0, R6, 0
            JSR    FACT          ; fact(n-1)

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

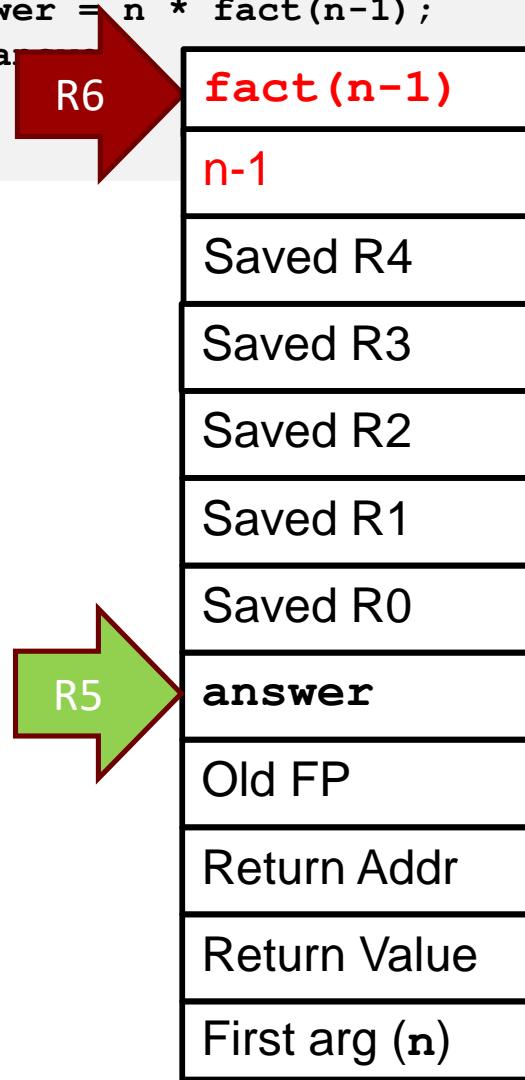
;     answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR  R0, R6, 0
          JSR  FACT           ; fact(n-1)
          LDR  R0, R6, 0 ; R0 = rv

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

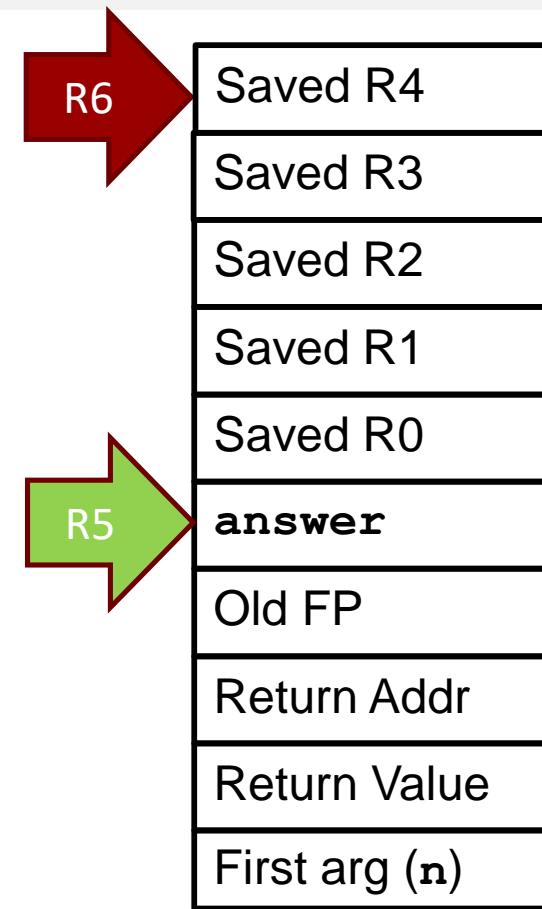
;      answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR   R0, R6, 0
          JSR   FACT           ; fact(n-1)
          LDR   R0, R6, 0 ; R0 = rv
          ADD   R6, R6, 2 ; Pop rv and arg1

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

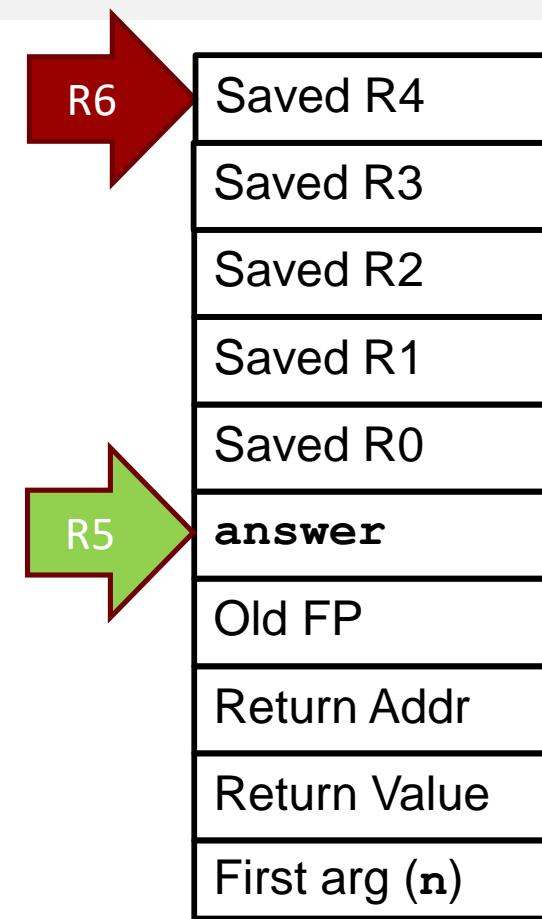
;      answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR   R0, R6, 0
          JSR   FACT           ; fact(n-1)
          LDR   R0, R6, 0 ; R0 = rv
          ADD   R6, R6, 2 ; Pop rv and arg1
;
answer = mult(n, R0)

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

;      answer = n * fact(n-1);

; rewrite: R0 = fact(n-1);answer = mult(n, R0)

IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR   R0, R6, 0
          JSR   FACT           ; fact(n-1)
          LDR   R0, R6, 0 ; R0 = rv
          ADD   R6, R6, 2 ; Pop rv and arg1

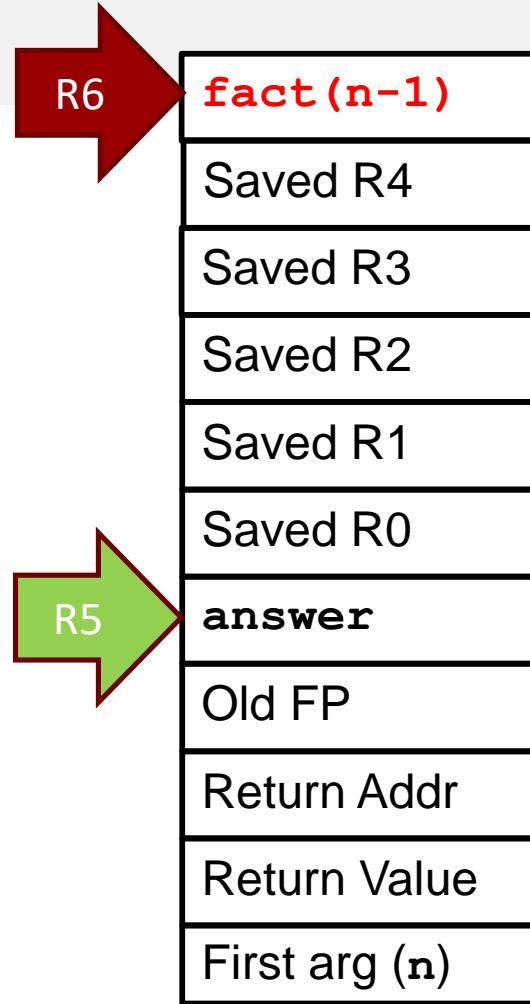
;      answer = mult(n, R0)
          ADD   R6, R6, -1; Push R0
          STR   R0, R6, 0

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

;     answer = n * fact(n-1);
; rewrite: R0 = fact(n-1);answer = mult(n, R0)
IFELSE1   LDR   R0, R5, 4 ; Push n-1
          ADD   R0, R0, -1
          ADD   R6, R6, -1
          STR   R0, R6, 0
          JSR   FACT      ; fact(n-1)
          LDR   R0, R6, 0 ; R0 = rv
          ADD   R6, R6, 2 ; Pop rv and arg1
;
          answer = mult(n, R0)
          ADD   R6, R6, -1; Push R0
          STR   R0, R6, 0
          ADD   R6, R6, -1; Push n
          LDR   R0, R5, 4
          STR   R0, R6, 0

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return a
}

```

R6	n
	fact(n-1)
	Saved R4
	Saved R3
	Saved R2
	Saved R1
	Saved R0
R5	answer
	Old FP
	Return Addr
	Return Value
	First arg (n)

```

; answer = mult(n, R0)
    ADD  R6, R6, -1; Push R0
    STR  R0, R6, 0
    ADD  R6, R6, -1; Push n
    LDR  R0, R5, 4
    STR  R0, R6, 0
    JSR  MULT      ; mult(n, R0)

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = R6 * fact(n-1);
    return answer;
}

```

n*fact(n-1)

n

fact(n-1)

Saved R4

Saved R3

Saved R2

Saved R1

Saved R0

answer

Old FP

Return Addr

Return Value

First arg (**n**)



R5

```

; answer = mult(n, R0)
    ADD  R6, R6, -1; Push R0
    STR  R0, R6, 0
    ADD  R6, R6, -1; Push n
    LDR  R0, R5, 4
    STR  R0, R6, 0
    JSR  MULT          ; mult(n, R0)
    LDR  R0, R6, 0 ; answer = rv
    STR  R0, R5, 0 ;

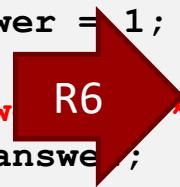
```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = R6 * fact(n-1);
    return answer;
}

```

n * fact(n-1)
n
fact(n-1)
Saved R4
Saved R3
Saved R2
Saved R1
Saved R0
answer
Old FP
Return Addr
Return Value
First arg (n)



```

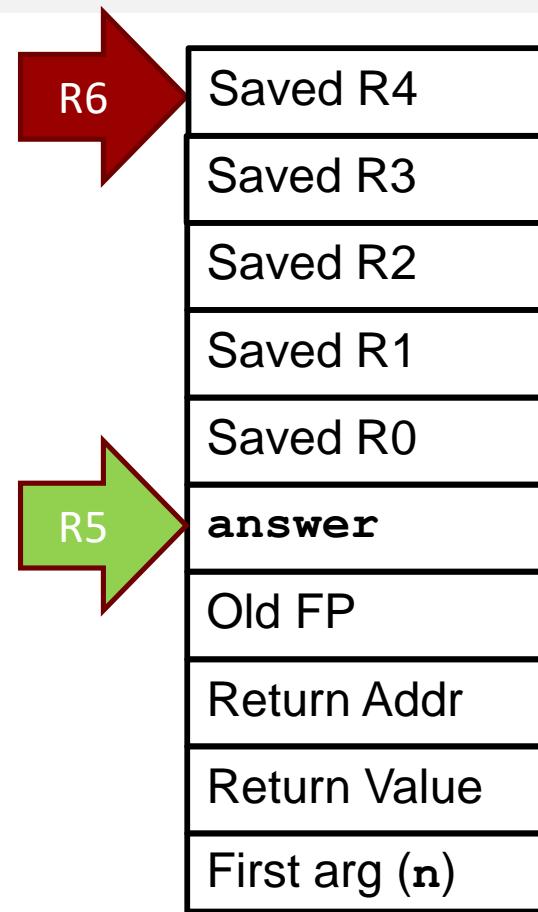
; answer = mult(n, R0)
    ADD  R6, R6, -1; Push R0
    STR  R0, R6, 0
    ADD  R6, R6, -1; Push n
    LDR  R0, R5, 4
    STR  R0, R6, 0
    JSR  MULT          ; mult(n, R0)
    LDR  R0, R6, 0 ; answer = rv
    STR  R0, R5, 0 ;
    ADD  R6, R6, 3 ; Pop rv and arg1-2

```

```

int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```



```

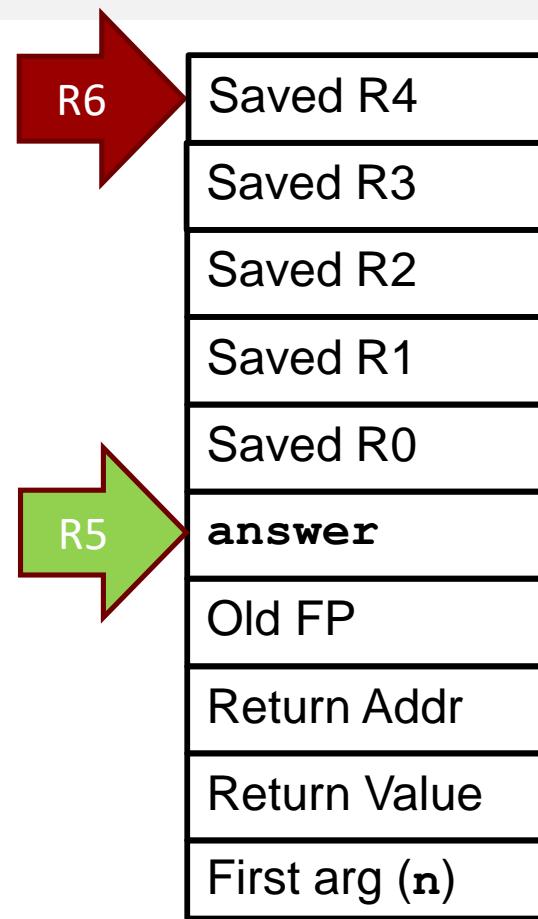
; answer = mult(n, R0)
    ADD  R6, R6, -1; Push R0
    STR  R0, R6, 0
    ADD  R6, R6, -1; Push n
    LDR  R0, R5, 4
    STR  R0, R6, 0
    JSR  MULT          ; mult(n, R0)
    LDR  R0, R6, 0 ; answer = rv
    STR  R0, R5, 0 ;
    ADD  R6, R6, 3 ; Pop rv and arg1-2
ENDIF1    NOP
;
; Tear down stack frame template goes below

```

```

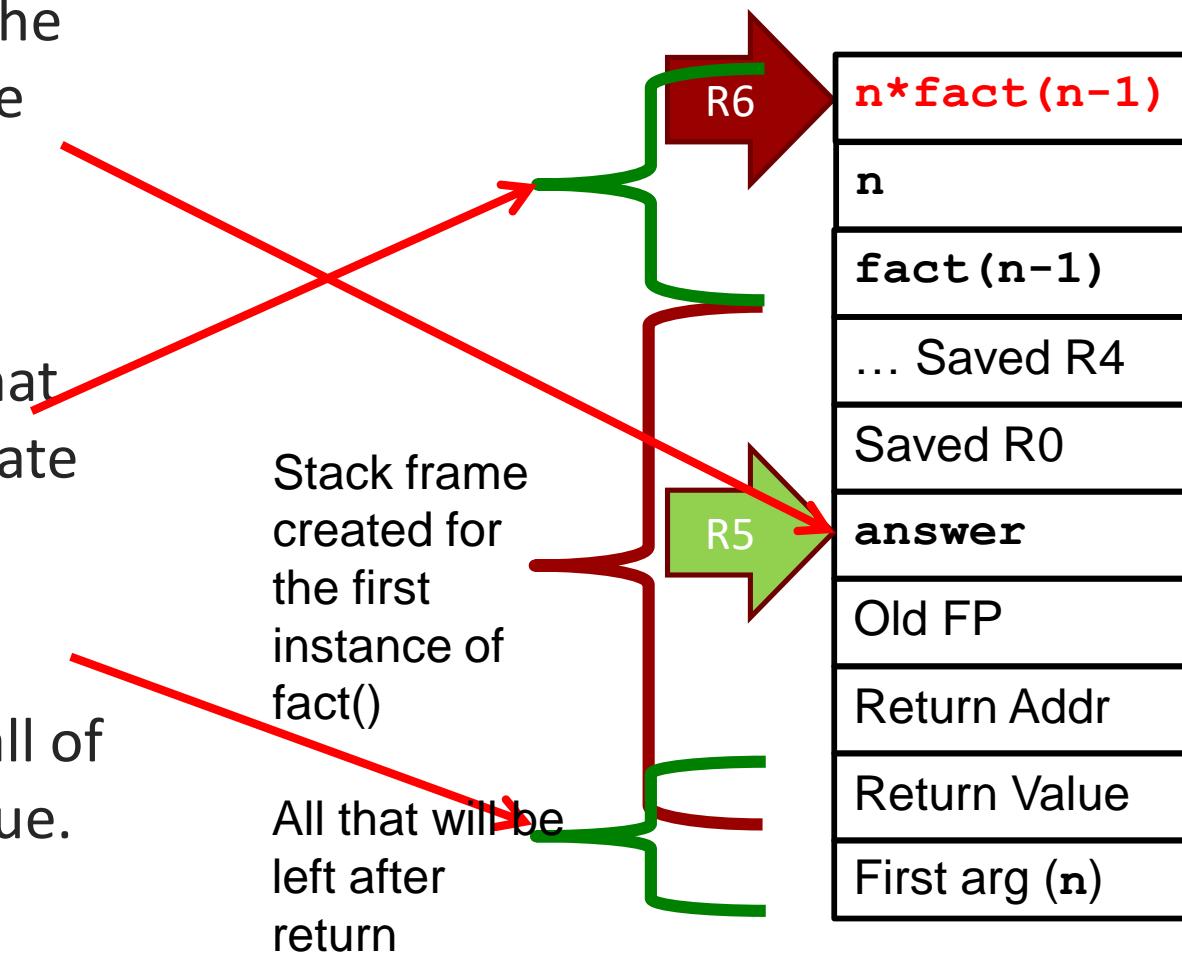
int fact(int n) {
    int answer;
    if(n <= 0)
        answer = 1;
    else
        answer = n * fact(n-1);
    return answer;
}

```

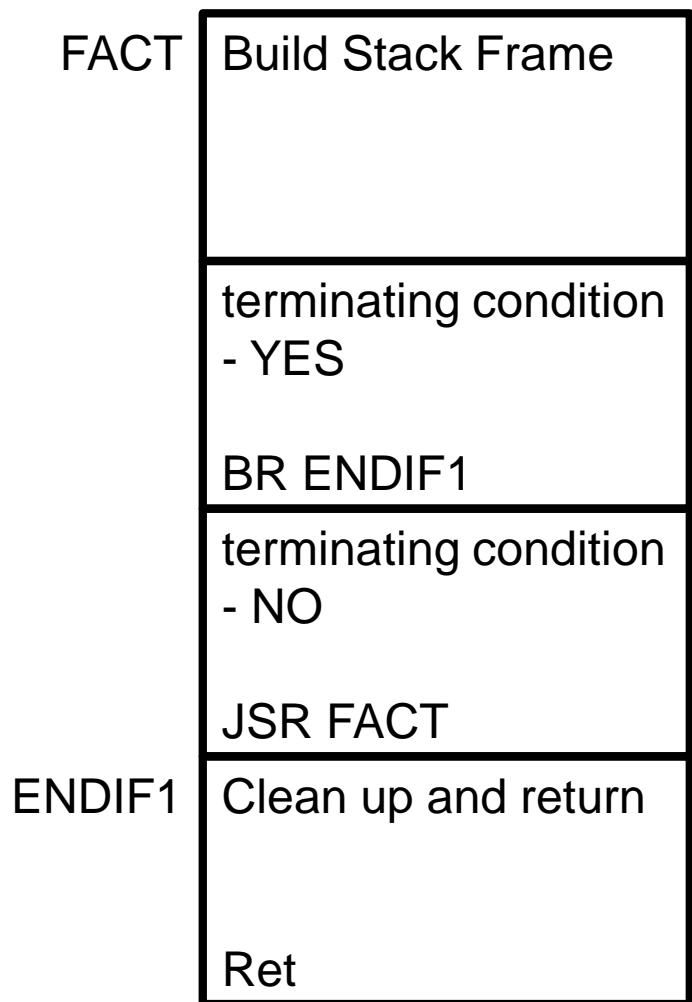


Some Things to Notice in fact()

- ↗ Note that in the stack frame created by the **beginning template code** for fact(), in the function body we only touched the local variables (a.k.a answer).
- ↗ All the rest of our stack use was above that fixed stack frame and used for intermediate computations.
- ↗ When fact() returns to its caller, the **ending template code** will throw away all of fact()'s stack frame **except** the return value.

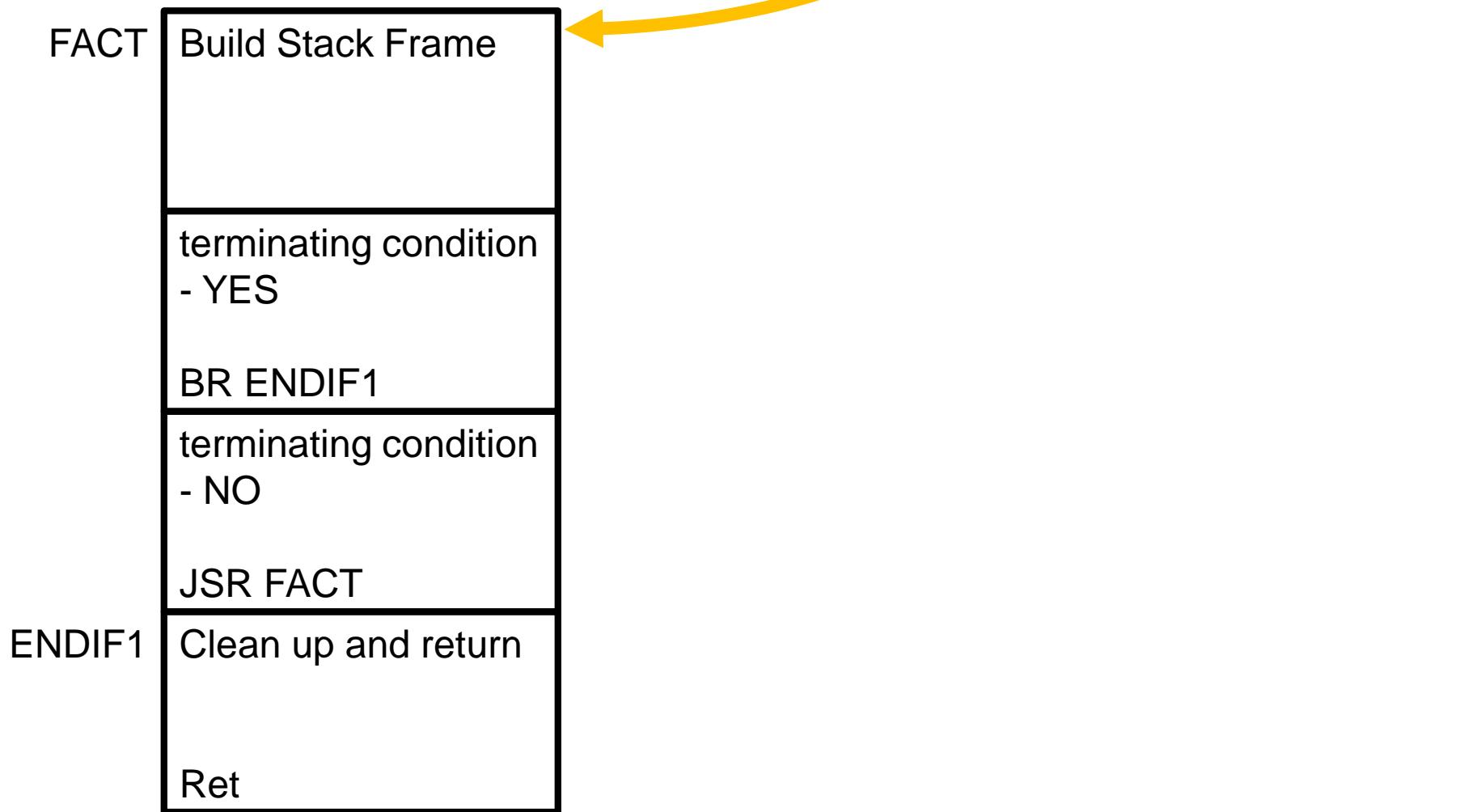


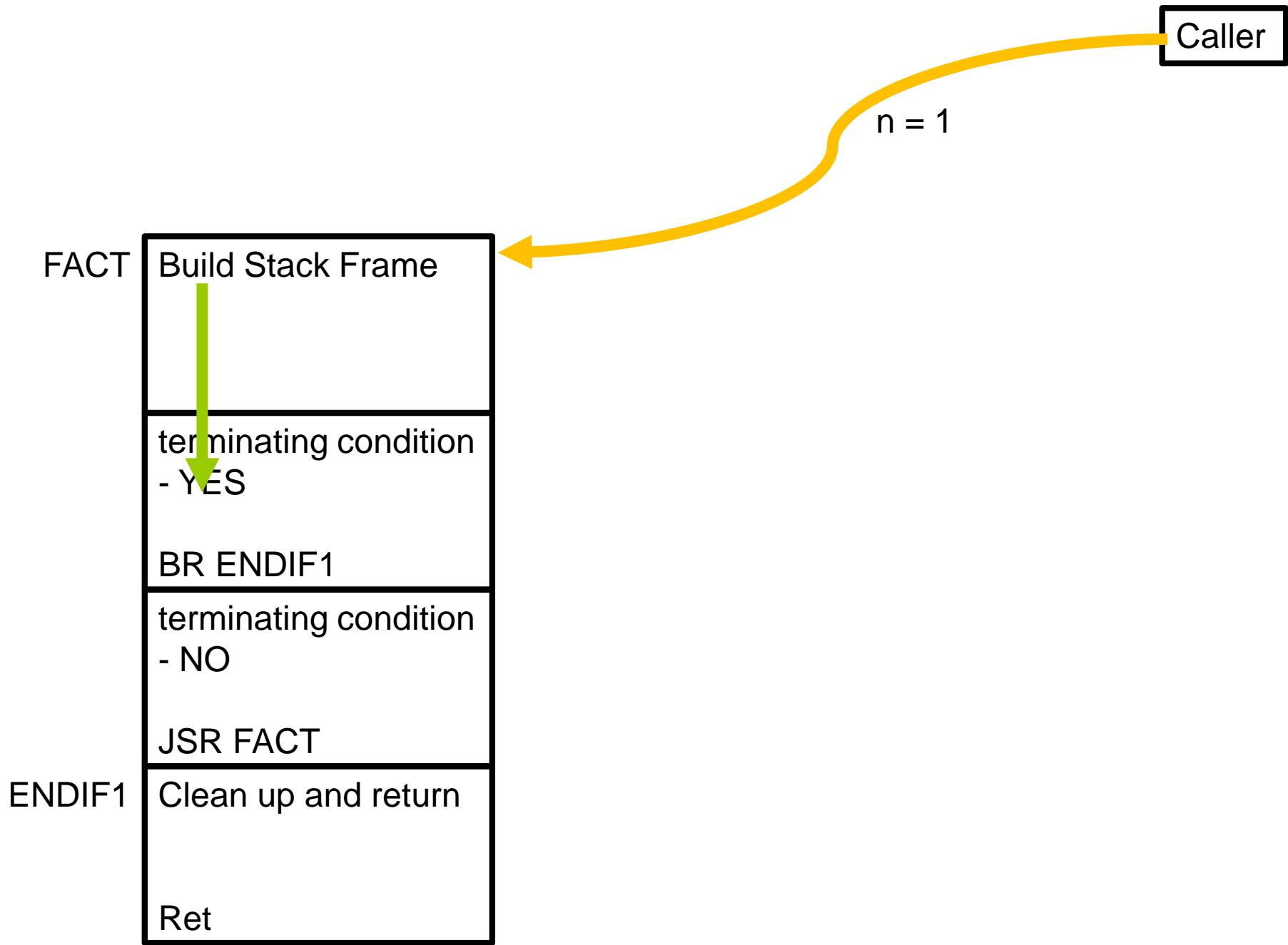
Big Picture

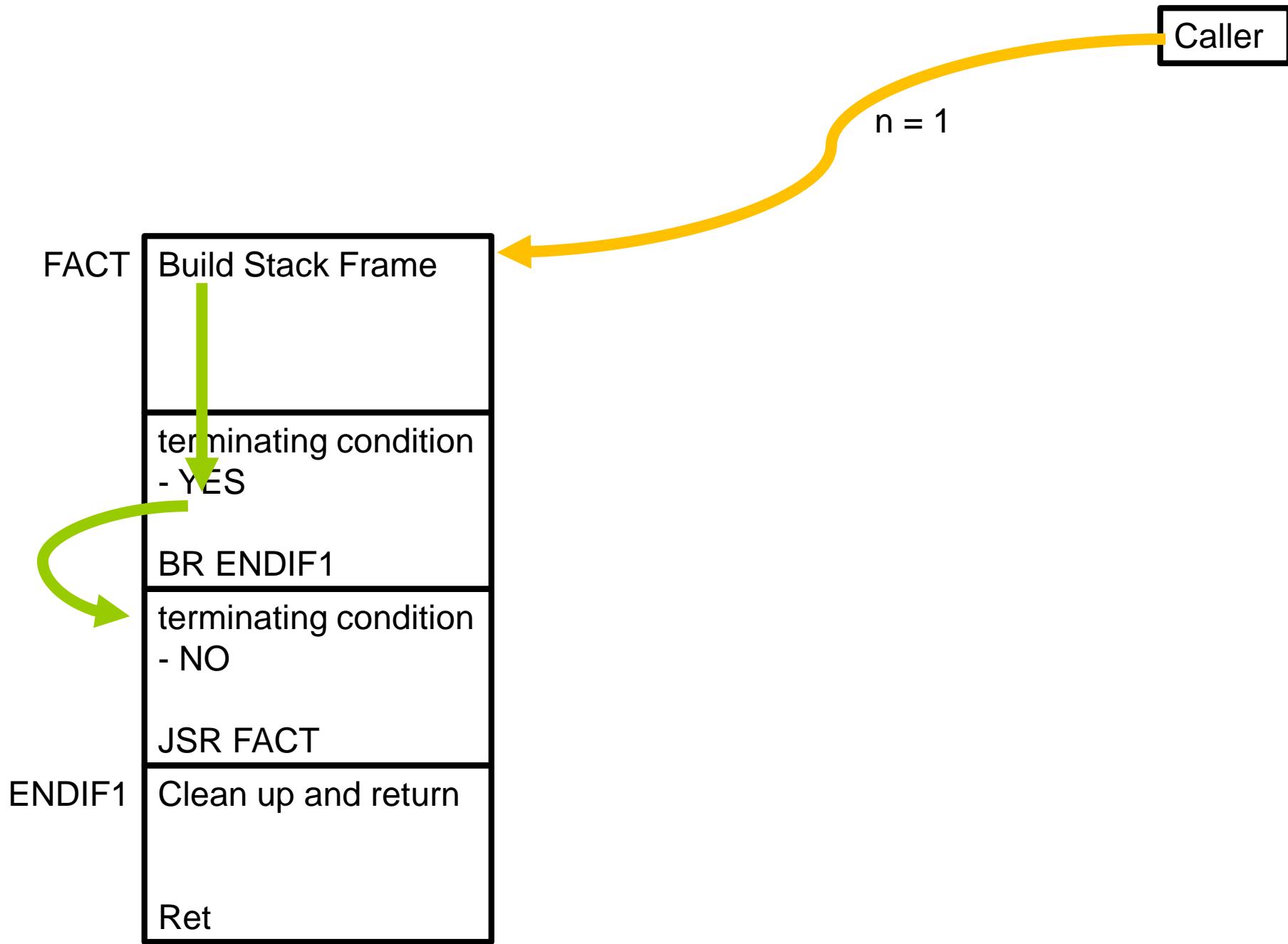


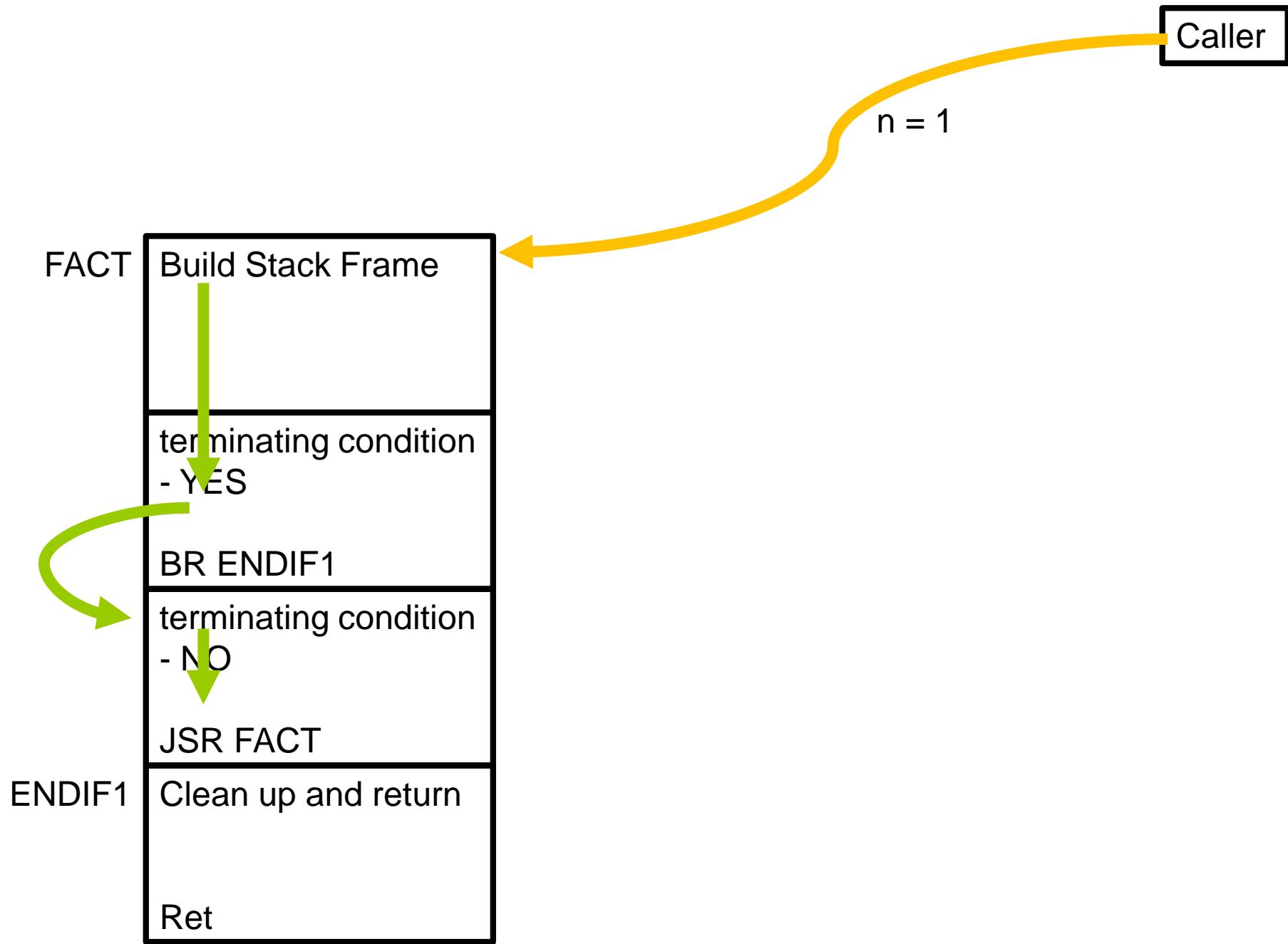
Don't try to memorize this control transfer. Let the abstraction of recursion cover that up for you!

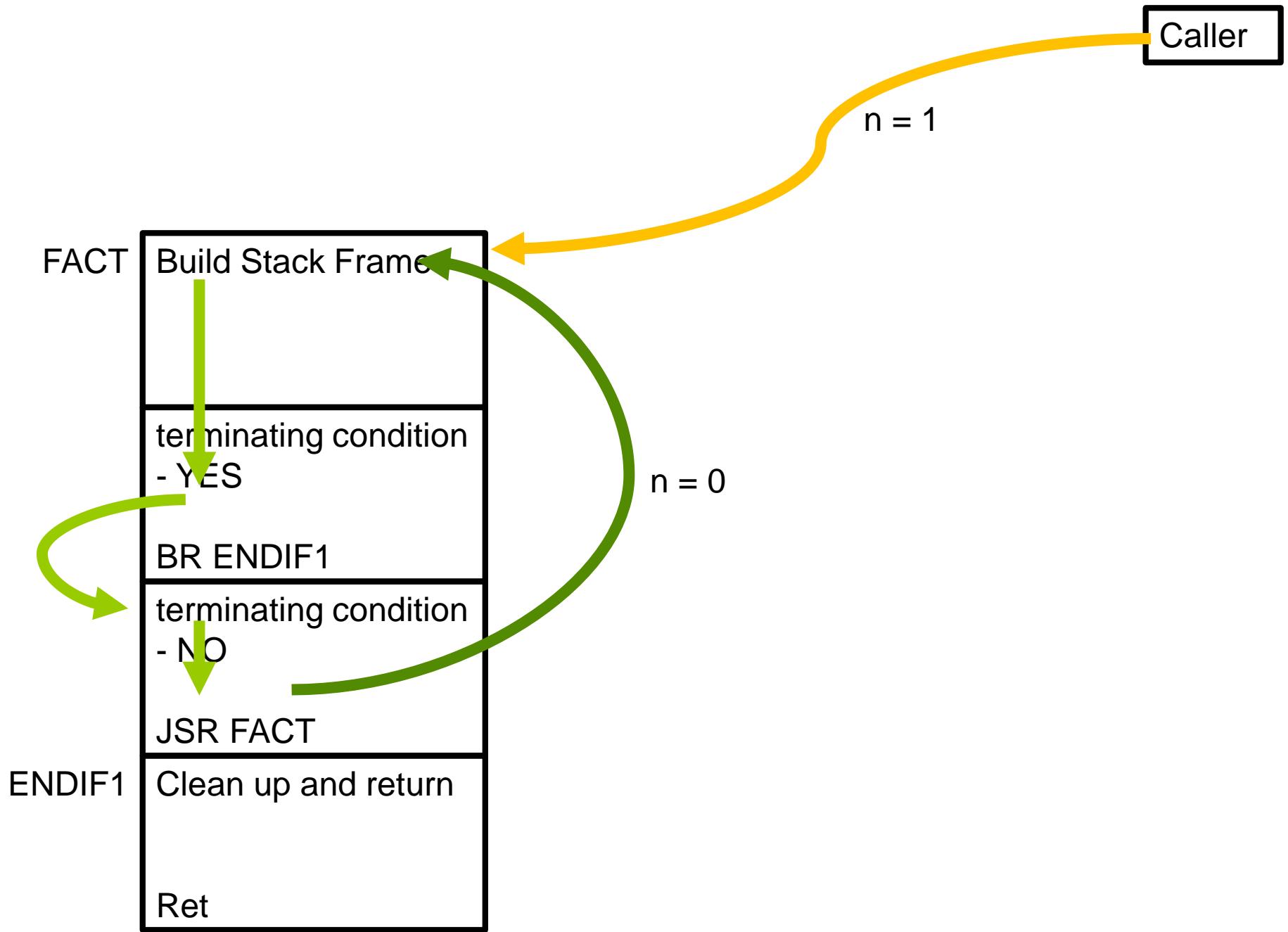
Big Picture

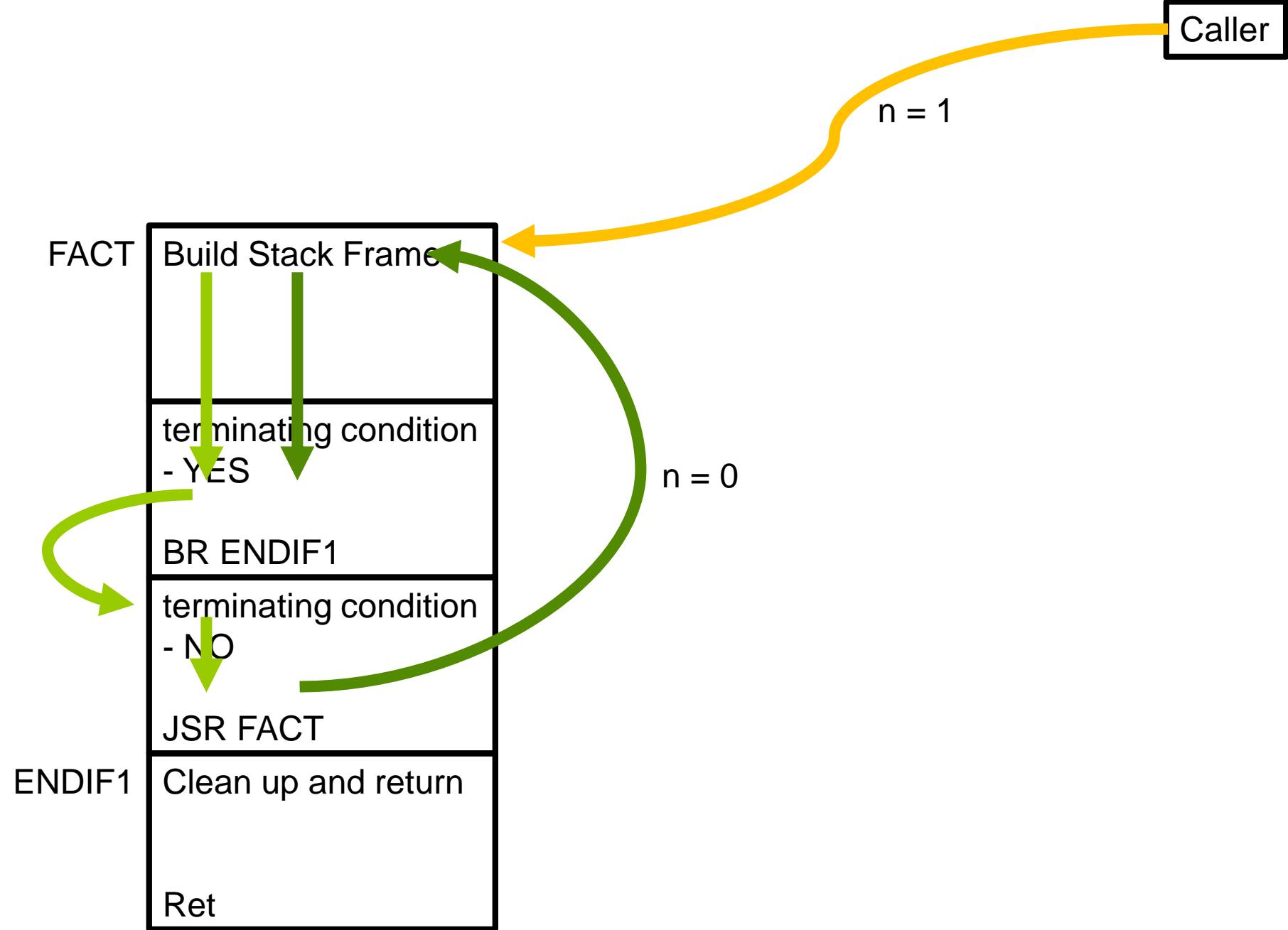


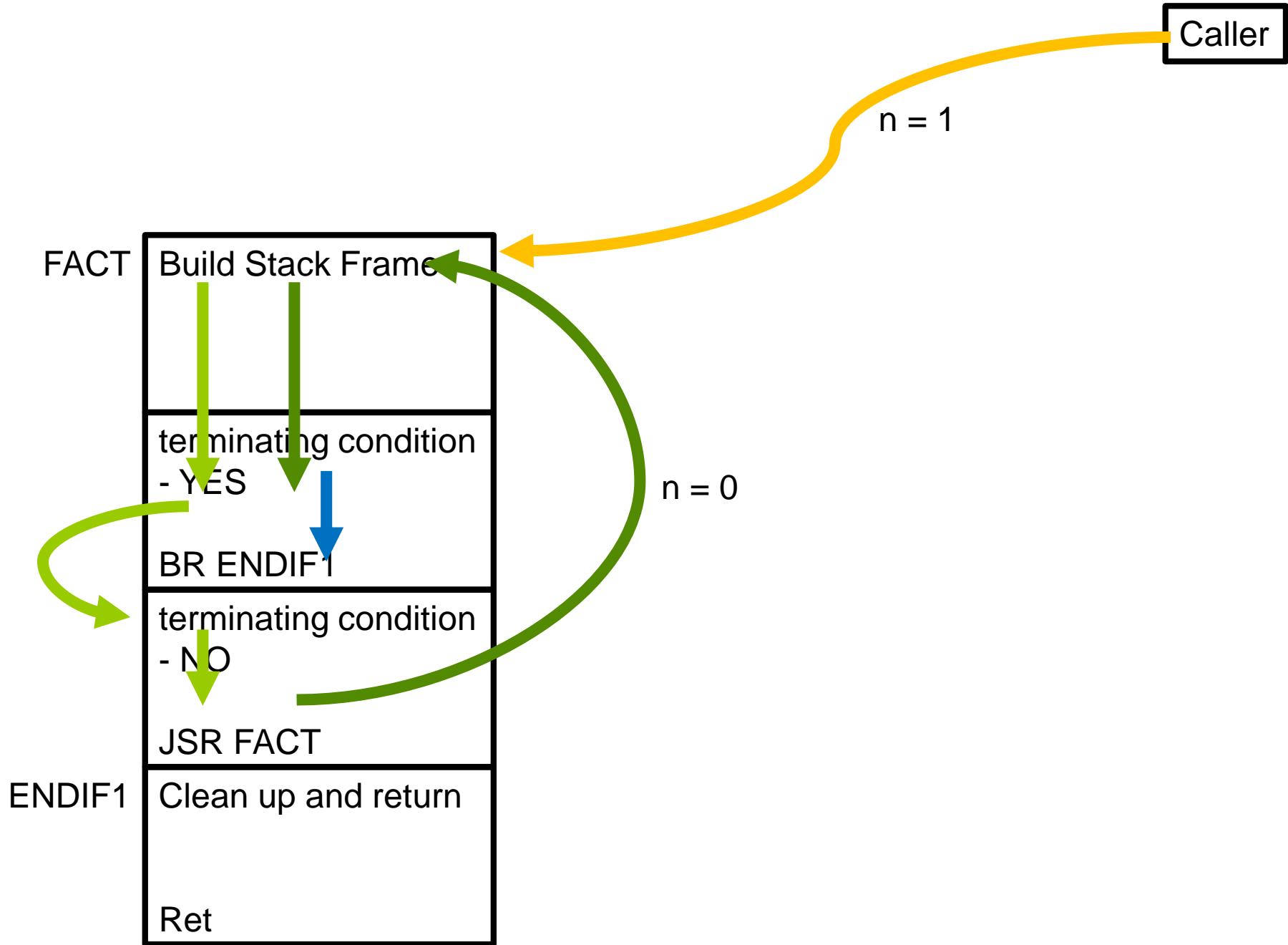


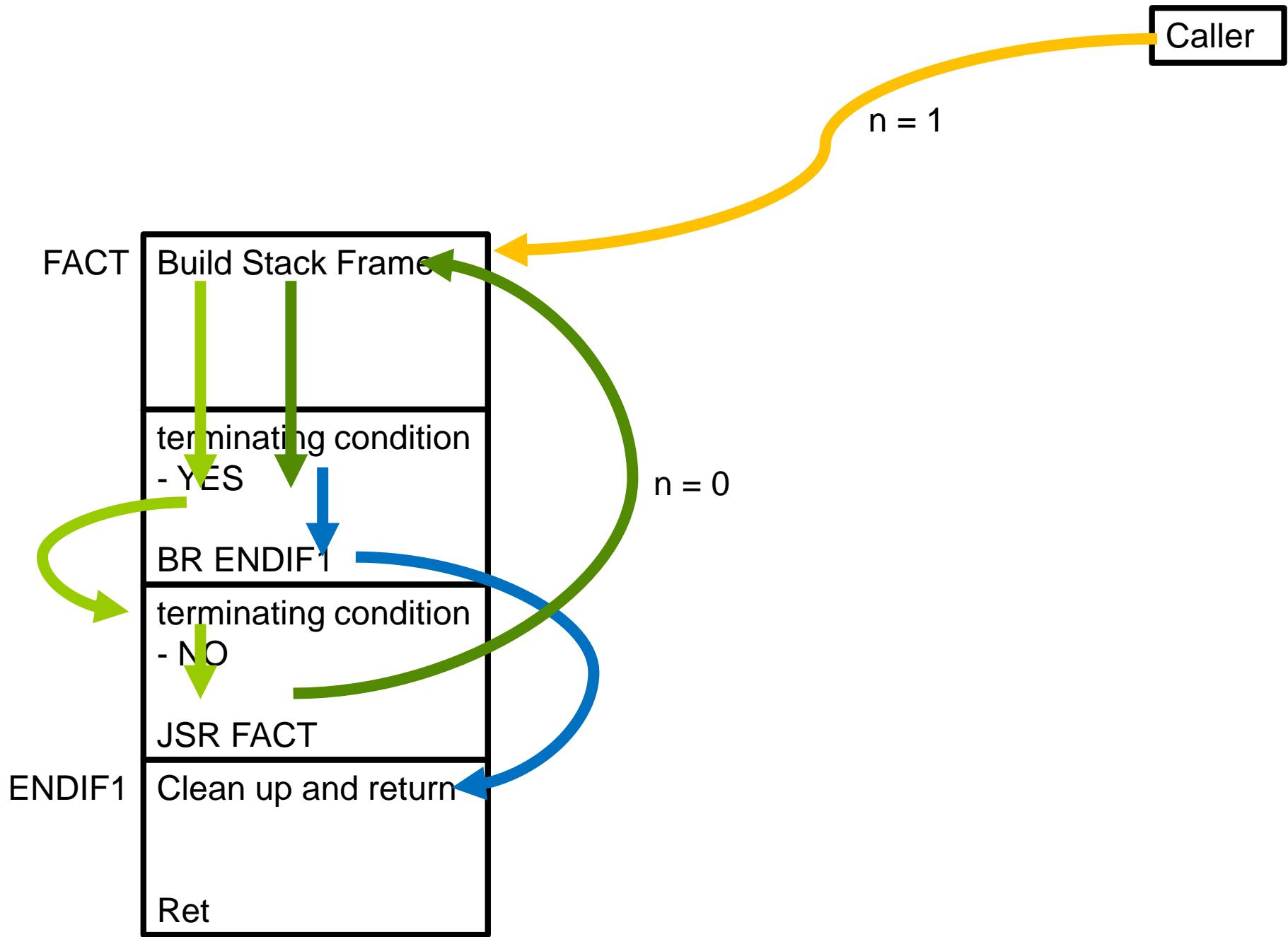


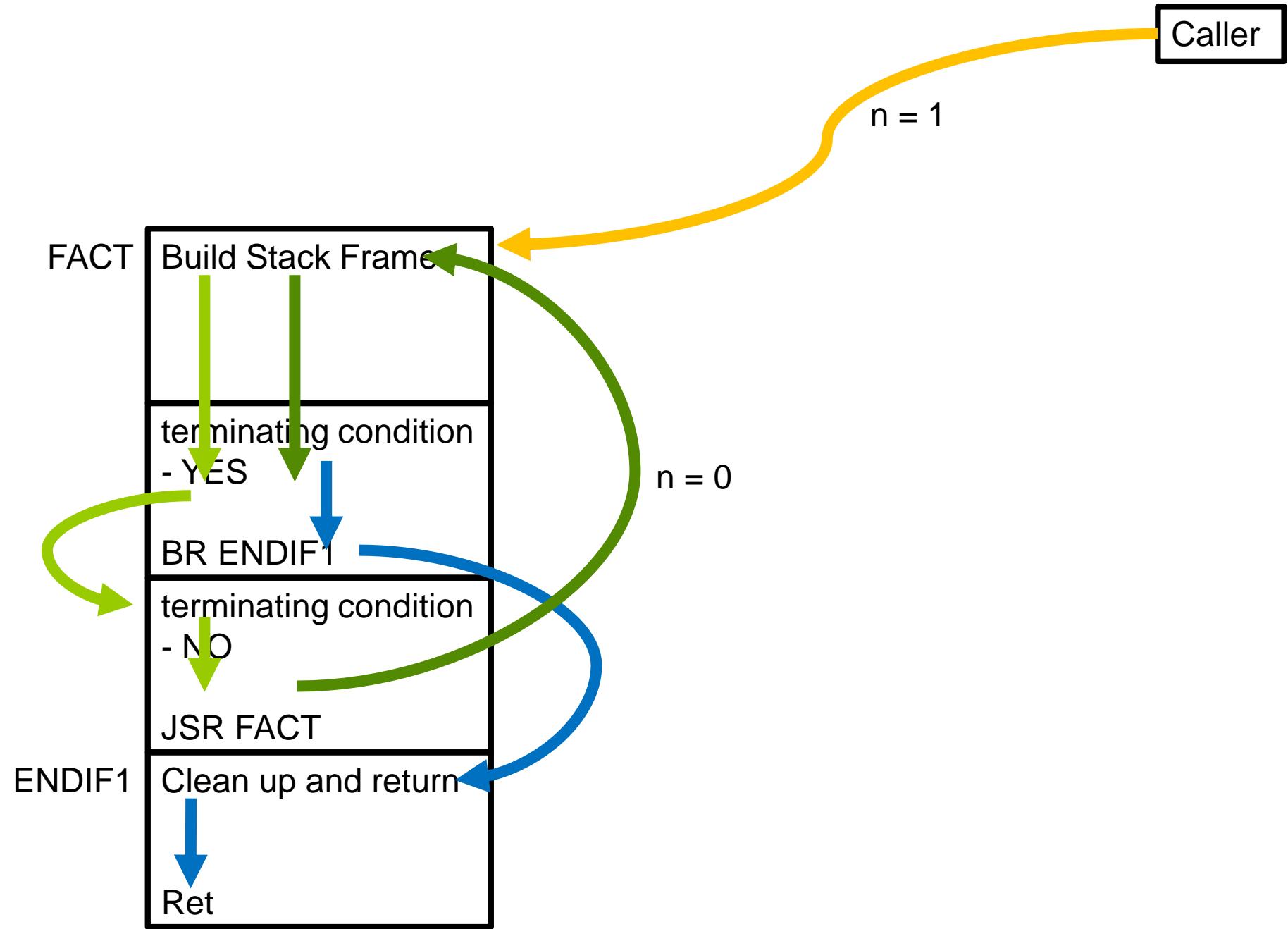


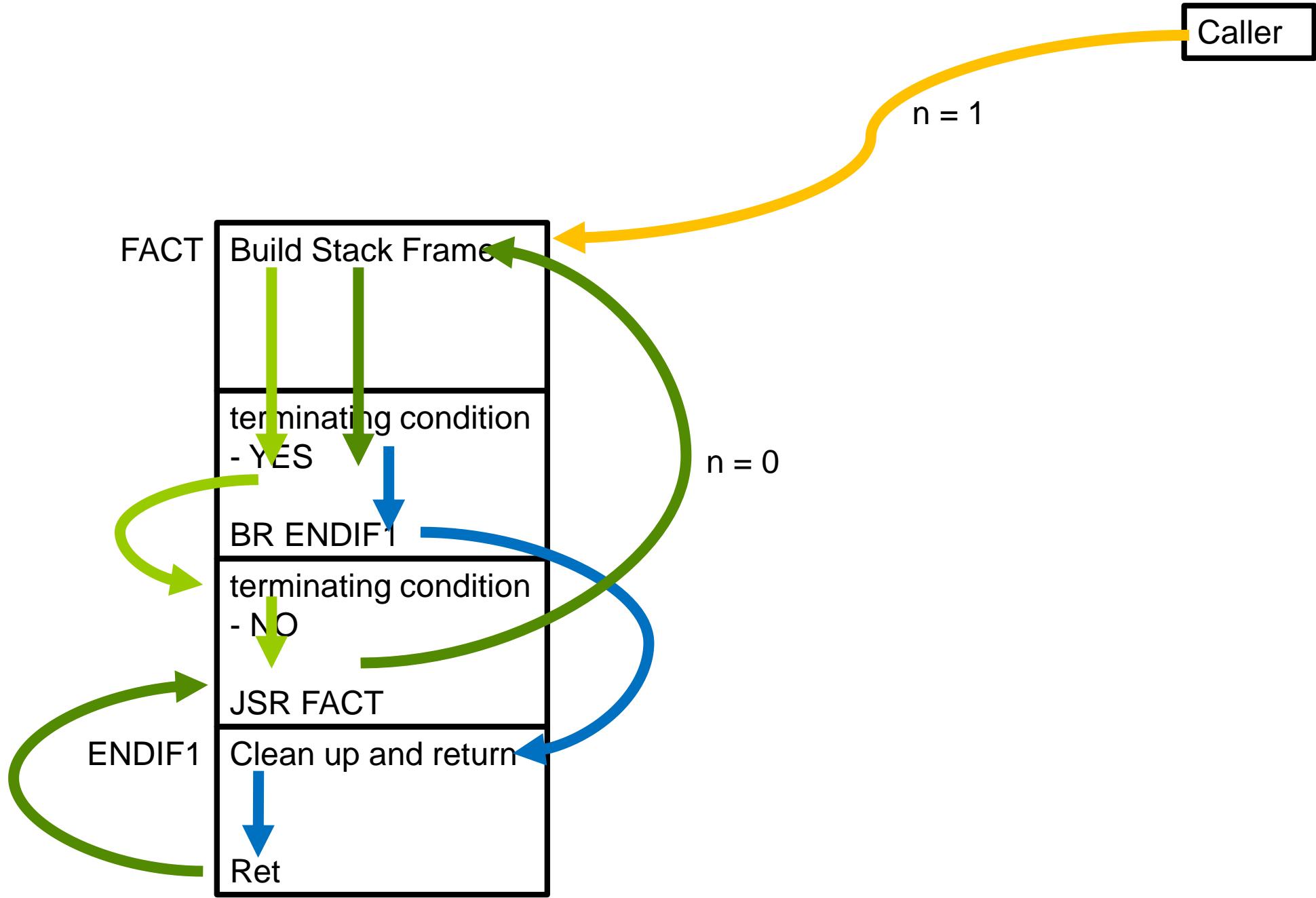


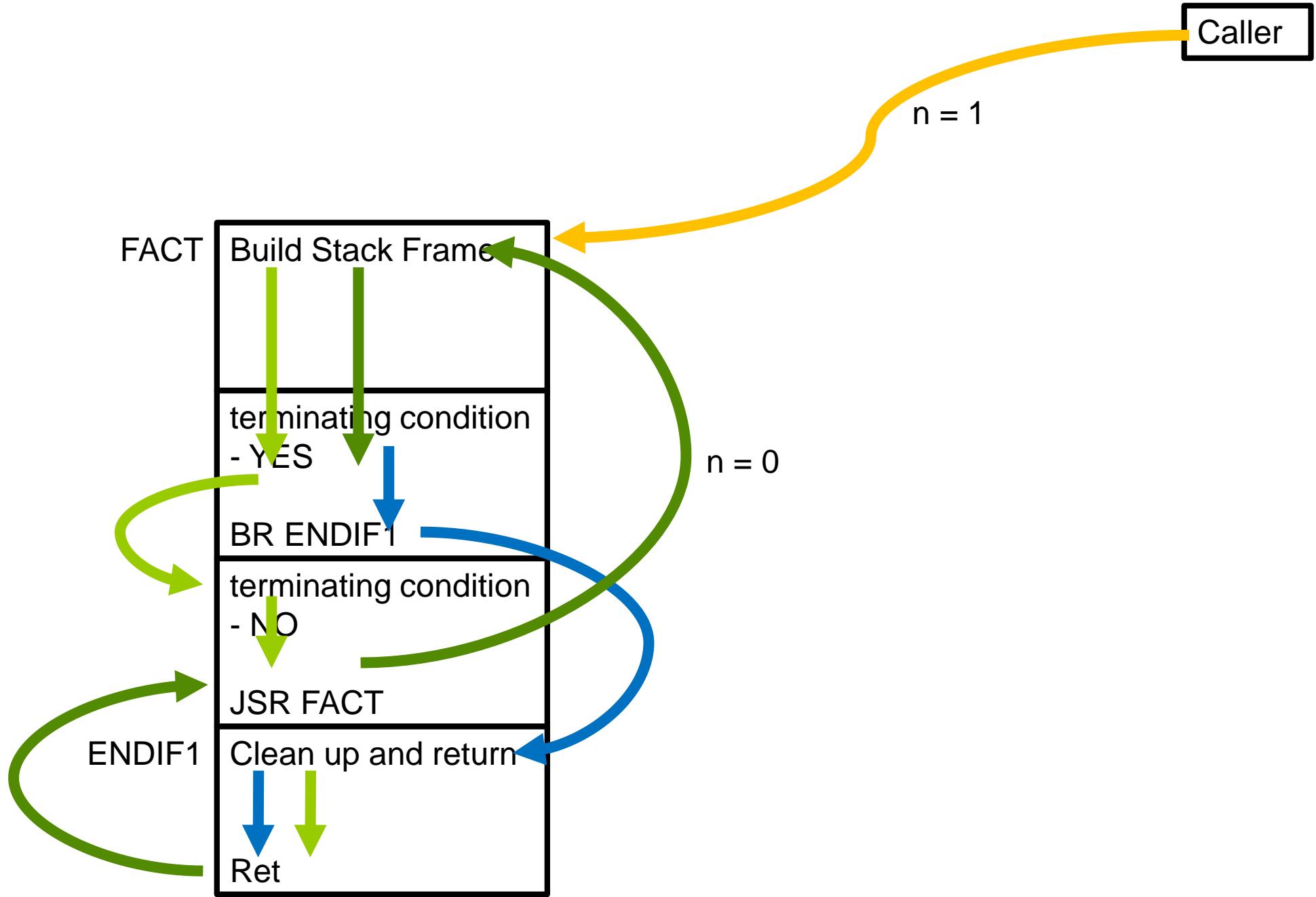


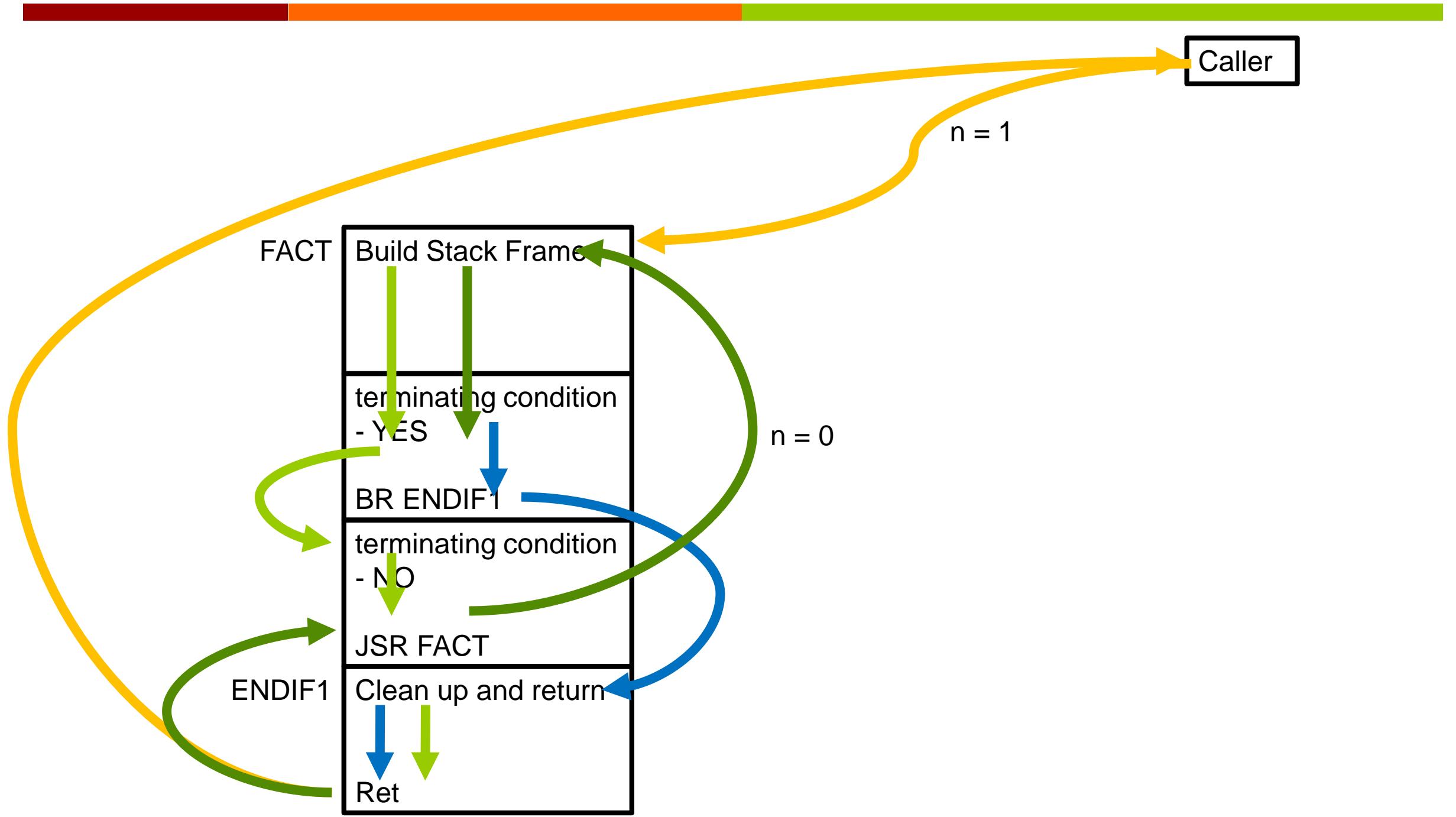




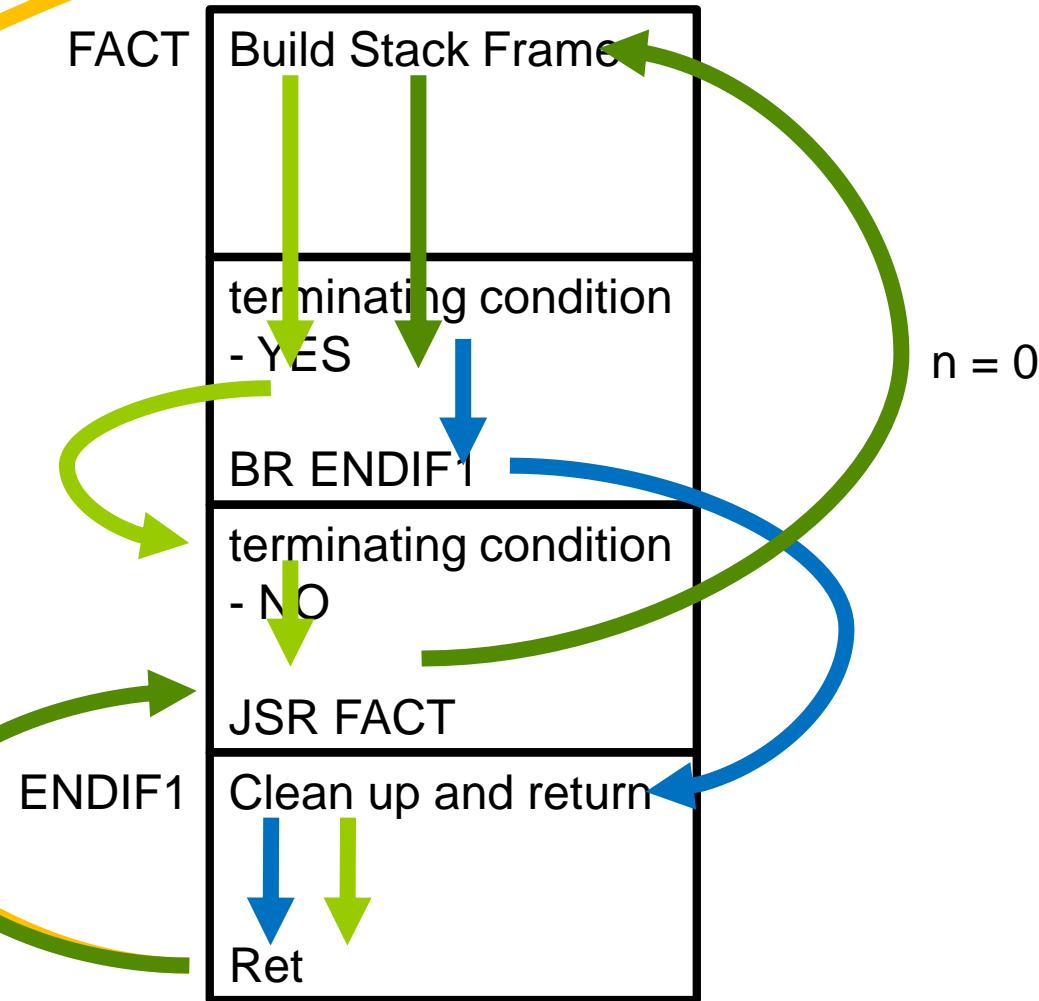








Caller



Note that we didn't have to account for any of this crazy control-transfer! The stack and the calling sequence handled it automatically!

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
         answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(2)

Trace Stack Frames: fact(2)

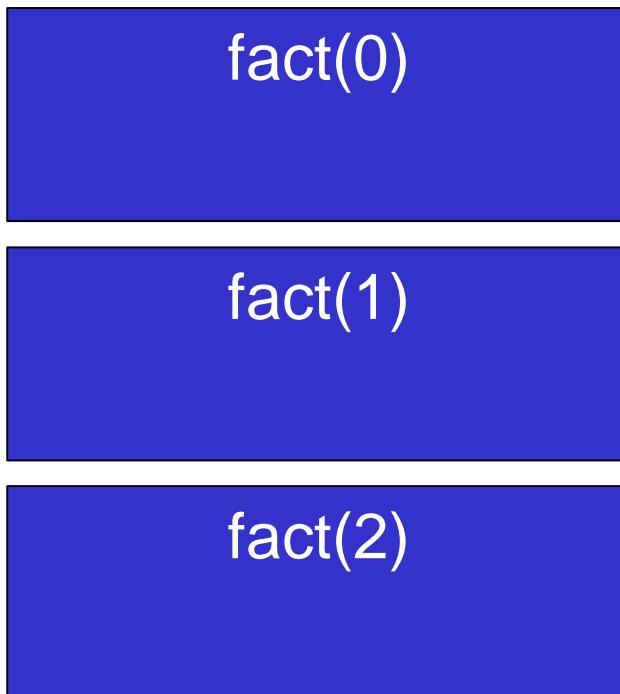
```
int fact(int n) {  
  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(1)



fact(2)

Trace Stack Frames: fact(2)



```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(0) returned: 1

fact(1)

fact(2)

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(1)

fact(2)



Trace Stack Frames: fact(2)

mult(1,1)

fact(1)

fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

Trace Stack Frames: fact(2)

mult(1,1) returned: 1

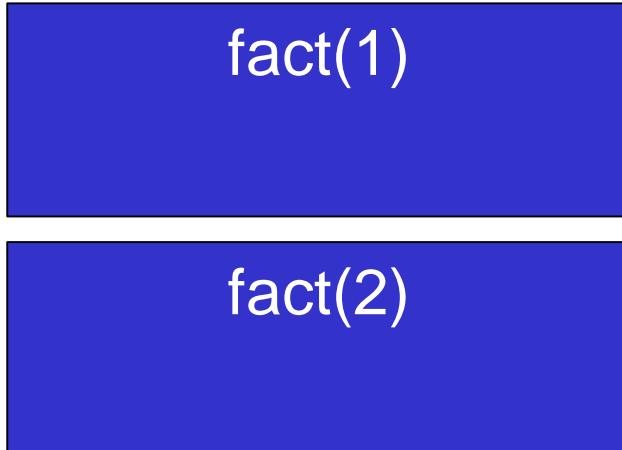
fact(1)

fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```



Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(1) returned: 1

fact(2)



Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
         answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(2)

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

mult(2,1)

fact(2)



Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

mult(2,1) returned: 2

fact(2)

Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
    return answer;  
}
```

fact(2)



Trace Stack Frames: fact(2)

```
int fact(int n) {  
    int answer;  
  
    if(n <= 0)  
        answer = 1;  
  
    else  
        answer = mult(n, fact(n-1));  
  
    return answer;  
}
```

fact(2) returned: 2

Conclusions?

- ↗ If you use a well-designed calling sequence, a recursive function call looks like any other function call.
- ↗ And a recursive function looks just like any other function.
- ↗ So why is merely the thought of writing recursive assembly code terrifying to some people?
- ↗ Good question.
- ↗ Please publish a paper if you figure it out. Lots of us would like to know the answer.

Questions?



Input/Output



- I/O Basics
 - Asynchronous vs. Synchronous
 - Memory Mapped I/O vs. Special I/O Instrs.
 - Interrupt Driven cs. Polling
 - LC-3 Implementation of Memory Mapped I/O
- Keyboard Input
- Monitor Output
- Full Implementation on LC-3

Synchronicity



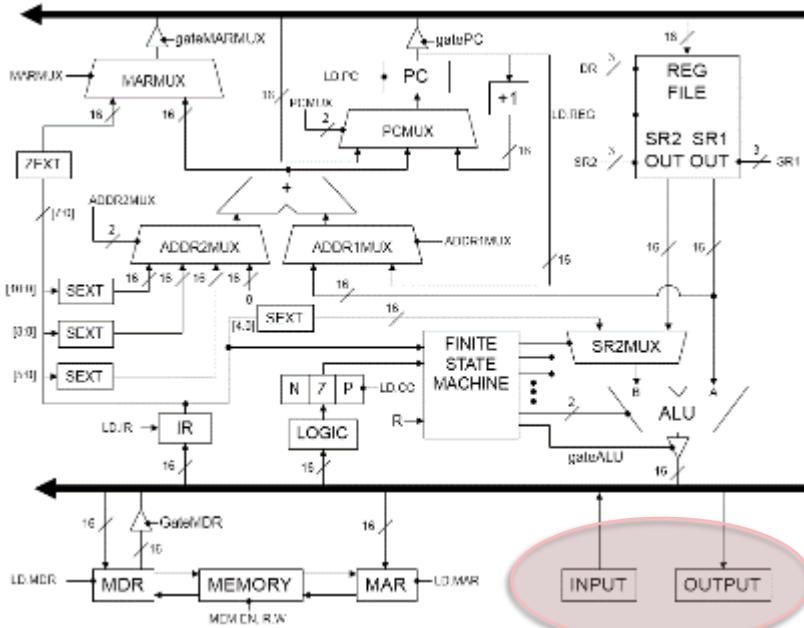
- ↗ Asynchronous
 - ↗ Electronic, mechanical and human speed
 - ↗ Speed mismatch
 - ↗ Handshaking: Ready bit

- ↗ Synchronous
 - ↗ Processor operation
 - ↗ Certain kinds of high speed I/O

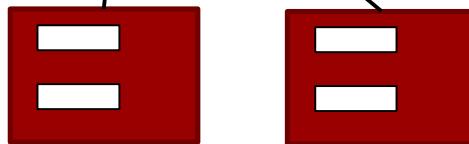
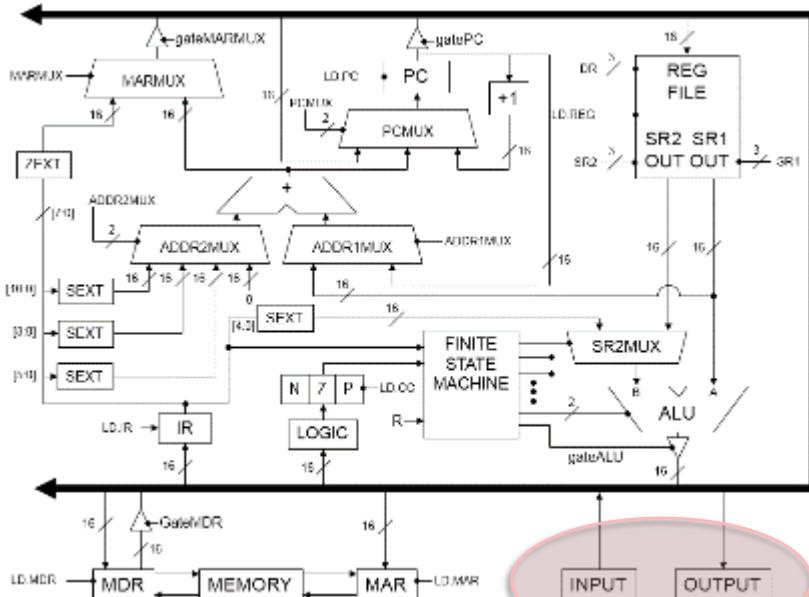
Transfer Timing

- I/O events generally happen much more slowly than CPU cycles.
- **Synchronous**
 - data supplied at a fixed, predictable rate
 - CPU reads/writes every X cycles
- **Asynchronous**
 - data rate less predictable
 - CPU must **synchronize** with device,
so that it doesn't miss data or write too quickly

How do we do I/O?



How we do I/O



Two Methods of Doing I/O

- Special I/O instructions
 - Need an opcode
 - Need to be general enough to work with devices that haven't been invented yet
- Memory-mapped I/O
 - Steal part of the address space for device registers
 - Operations done by reading/writing bits in the device registers
 - (We're going to concentrate on this method)

Device Registers

- ↗ **Data registers:** Used for the actual transfer of data i.e. character code
- ↗ **Status registers:** Information the device is telling us
- ↗ **Control registers:** Allows us to set changeable device characteristics

- ↗ Device registers are often part of the i/o device itself

Memory Mapped vs. Special I/O Instructions

- ☞ If device registers are located at valid memory addresses, how can we access them?
- ☞ How **can** they be located at valid memory addresses?
- ☞ "The very old PDP-8 (from DEC, light years ago—1965) is an example of a computer that used special I/O instructions."



Interrupt-Driven vs. Polling

- Two ways to handle I/O completion
- Interrupt-driven
 - Excuse me but I've just given you a character
- Polling
 - Has a character been typed?
 - (We'll start with this approach)

Family Vacation!

Dad, Dad, are we there yet?

Not yet.

Dad, Dad, are we there yet?

Not yet.

Dad, Dad, are we there yet?

Not yet. (sigh)

.....

This is polling, right?

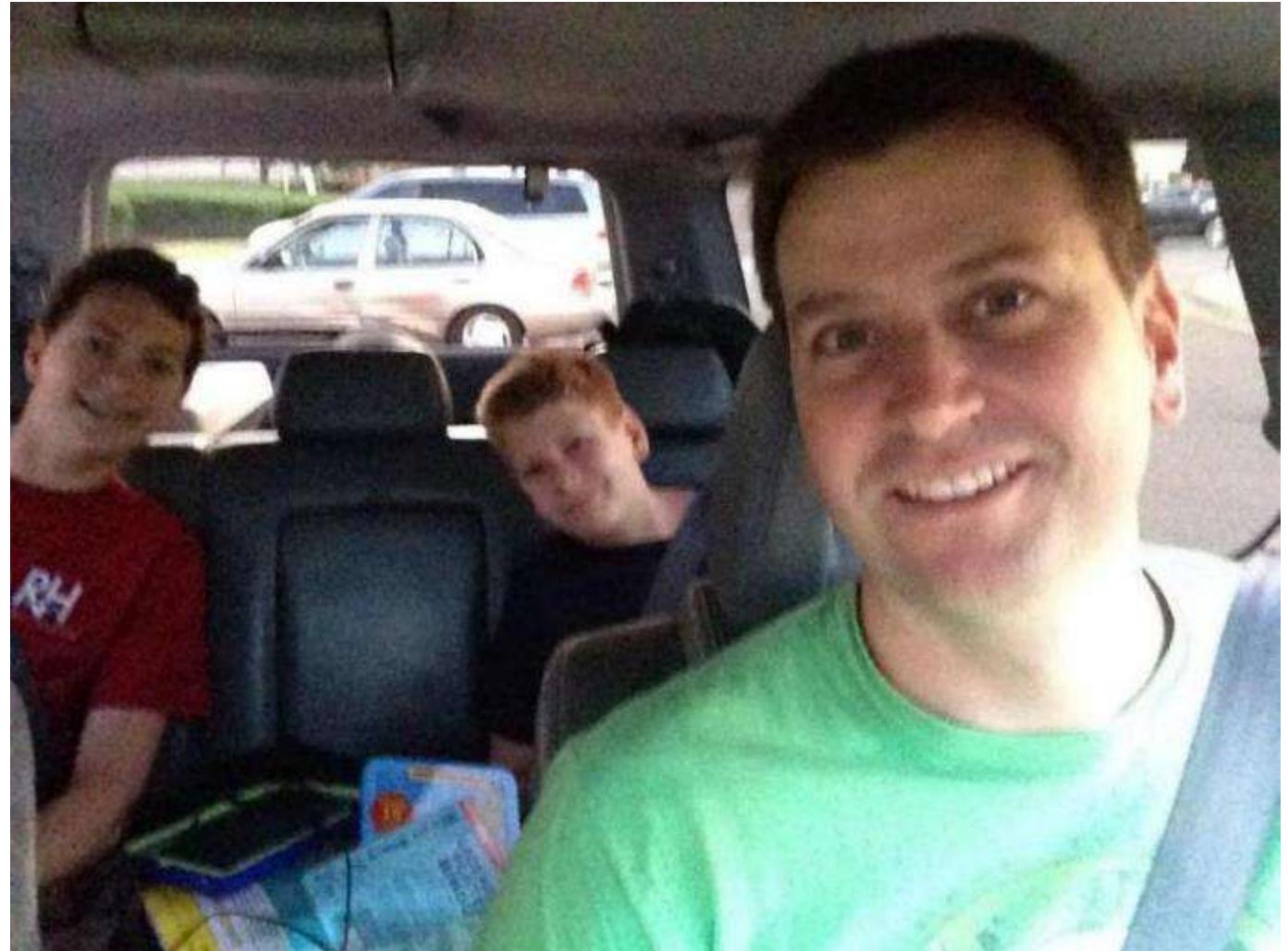
So what would the interrupt script sound like?

Dad, Dad, are we there yet?

No. Put your heads back, take a nap
and I'll wake you when we get there.

(long silence)

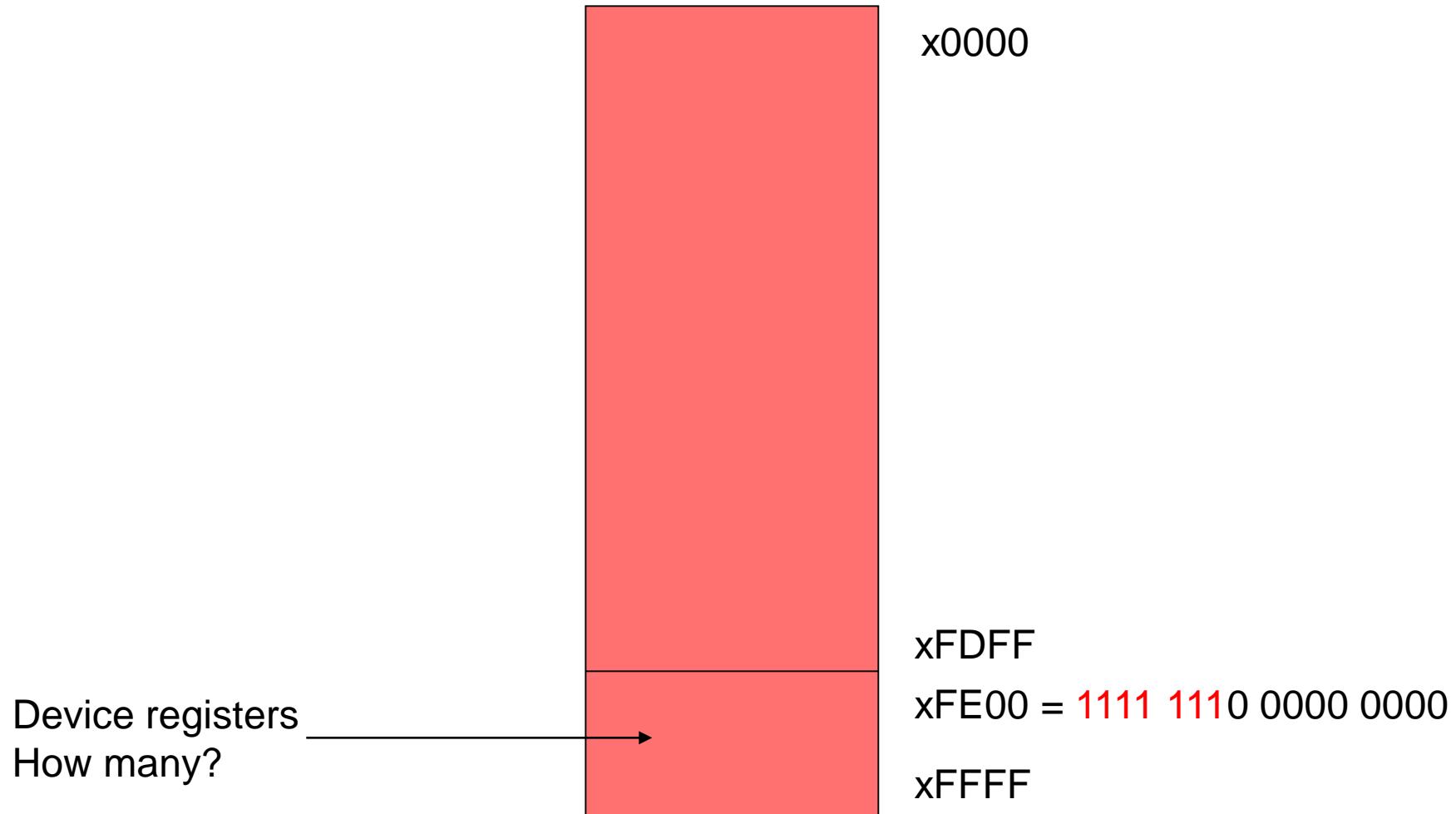
OK kids, wake up! We're here!



Our First Attempts at I/O

- ↗ We'll be working with **asynchronous I/O** instead of synchronous
- ↗ We're going to use **memory mapped I/O** instead of special I/O instructions
- ↗ We're going to use **polling** instead of interrupts

LC-3 Address Space



Keyboard Input

- ☞ KBSR (x_{FE00})



- ☞ Only uses 1 bit. Why bit 15 instead of 0?
- ☞ Bit 15 is set when a character is available

- ☞ KBDR (x_{FE02})

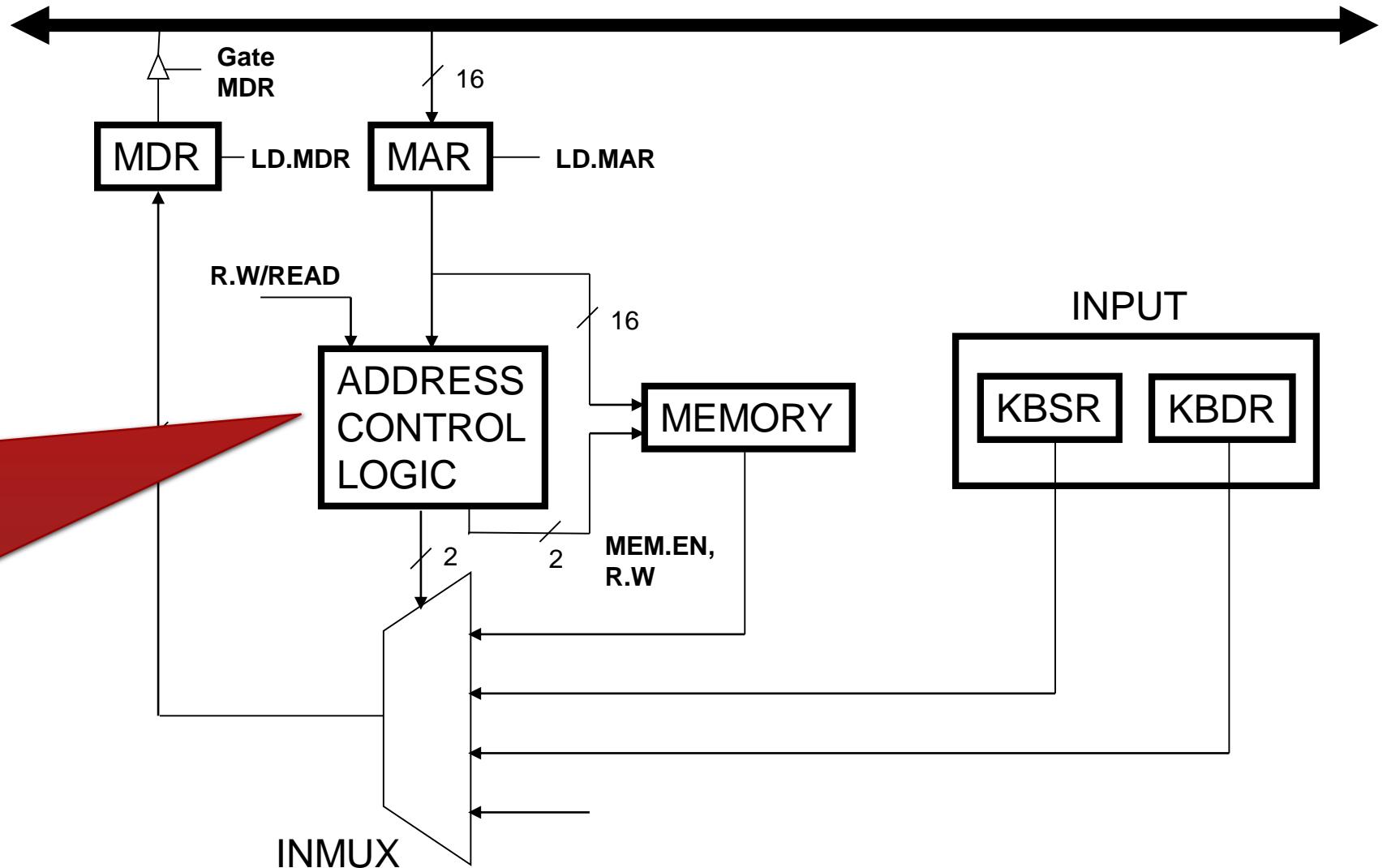


- ☞ Really only need 8 bits but 16 easier
- ☞ This location is read-only
- ☞ Reading clears KBSR
- ☞ (There's also an Interrupt Enable bit not shown in KBSR to indicate we want to be interrupted when KBSR[15] is set to 1, but we're going to wait to discuss that)

Memory Mapped Input

Can you build a circuit
that does this?

```
If MAR == 0xFE00    INMUX=1
else if MAR == 0xFE02   INMUX=2
else                  INMUX=0
```



; Read characters from the keyboard until CTRL/Z

```
.orig x3000
ld r4, term
lea r2, buffer ; Initialize buffer pointer
start ldi r1, kbsrA ; See if a char is there
BRzp start
ldi r0, kbdrA ; get the character
str r0, r2, 0 ; Store it in buffer
not r0, r0 ; subtract R4-R0
add r0, r0, #1 ; to check for termination
add r0, r0, r4 ; char stored in R4
brZ quit
add r2, r2, 1 ; Increment buffer pointer
br start ; Do it again!

quit halt
term .fill x001A ; CTRL/Z
kbsrA .fill xfe00
kbdrA .fill xfe02
buffer .blkw x0100
.end
```

Monitor Output

- ↗ DSR (x_{FE04})

A small diagram of the DSR register. It consists of a red rectangle divided into two equal-width horizontal sections. The left section is white and contains the text "rdy". The right section is also white.

- ↗ Transferring a character to DDR clears DSR
- ↗ When monitor is finished processing a character it sets DSR bit 15
- ↗ "Please sir, may I have another?"

- ↗ DDR (x_{FE06})

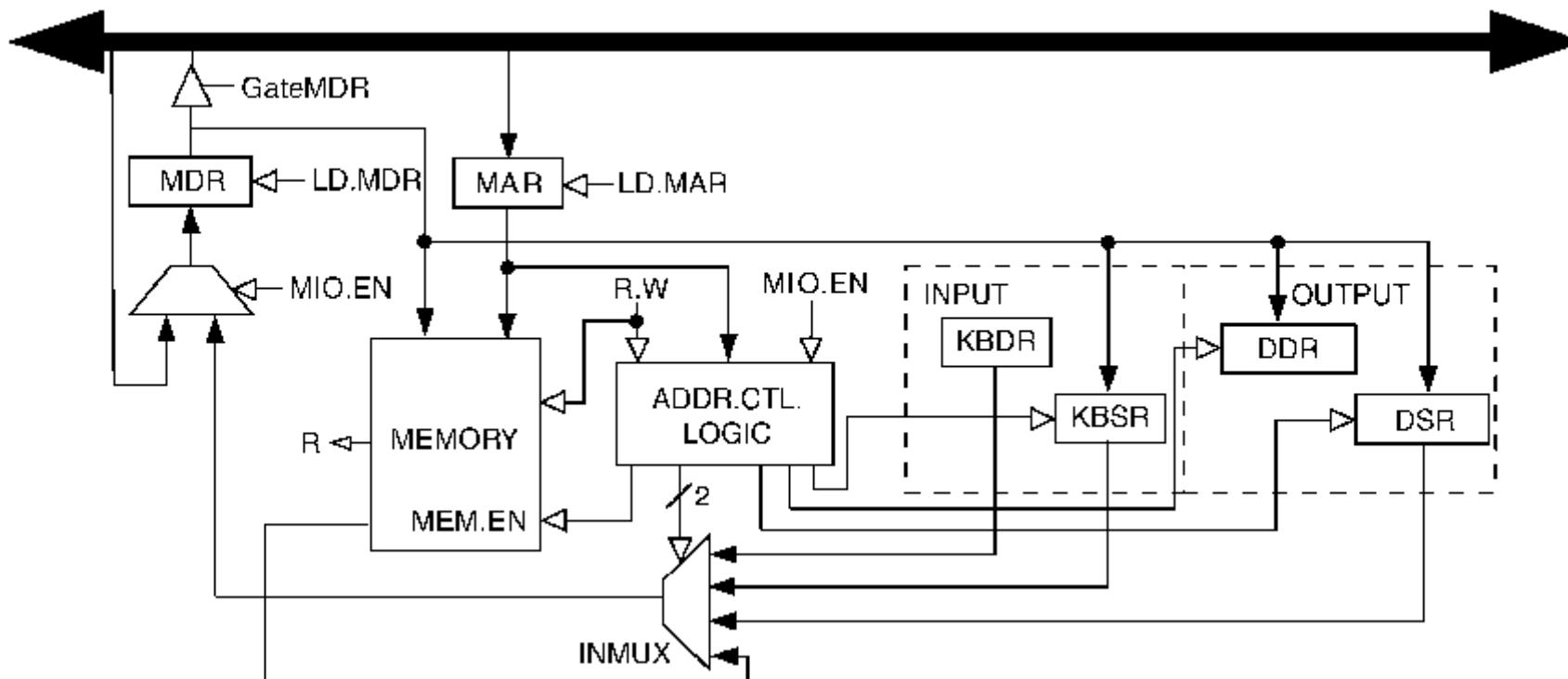
A small diagram of the DDR register. It consists of a red rectangle divided into two equal-width horizontal sections. The left section is red and the right section is white and contains the text "data".

- ↗ Transfer character to this address to print it on the monitor
- ↗ (There's also an Interrupt Enable bit not shown in DSR to indicate we want to be interrupted when DSR[15] is set to 1, but we're going to wait to discuss that)

; Write the contents of a string to the display

```
.orig x3000
       lea    r2, buffer ; Initialize buffer ptr
start   ldr    r0, r2, 0  ; Get char into r0
       brZ    quit      ; Terminate on null
wait    ldi    r3, dsrA ; Are we ready?
       brZP   wait
       sti    r0, ddrA ; Send R0 to monitor
       add    r2, r2, 1  ; Move buffer ptr over 1
       br    start
quit    halt
dsrA    .fill   xfe04
ddrA    .fill   xfe06
buffer  .stringz "Hello, World!"
       .end
```

Full Implementation of LC-3 Memory-Mapped I/O



Because of interrupt enable bits, status registers (KBSR/DSR) must be written, as well as read.

Questions?

- I/O Basics
 - Asynchronous vs. Synchronous
 - Memory Mapped I/O vs. Special I/O Instrs.
 - Interrupt Driven cs. Polling
 - LC-3 Implementation of Memory Mapped I/O
- Keyboard Input
- Monitor Output
- Full Implementation on LC-3

Interrupts, TRAPs, and Exceptions

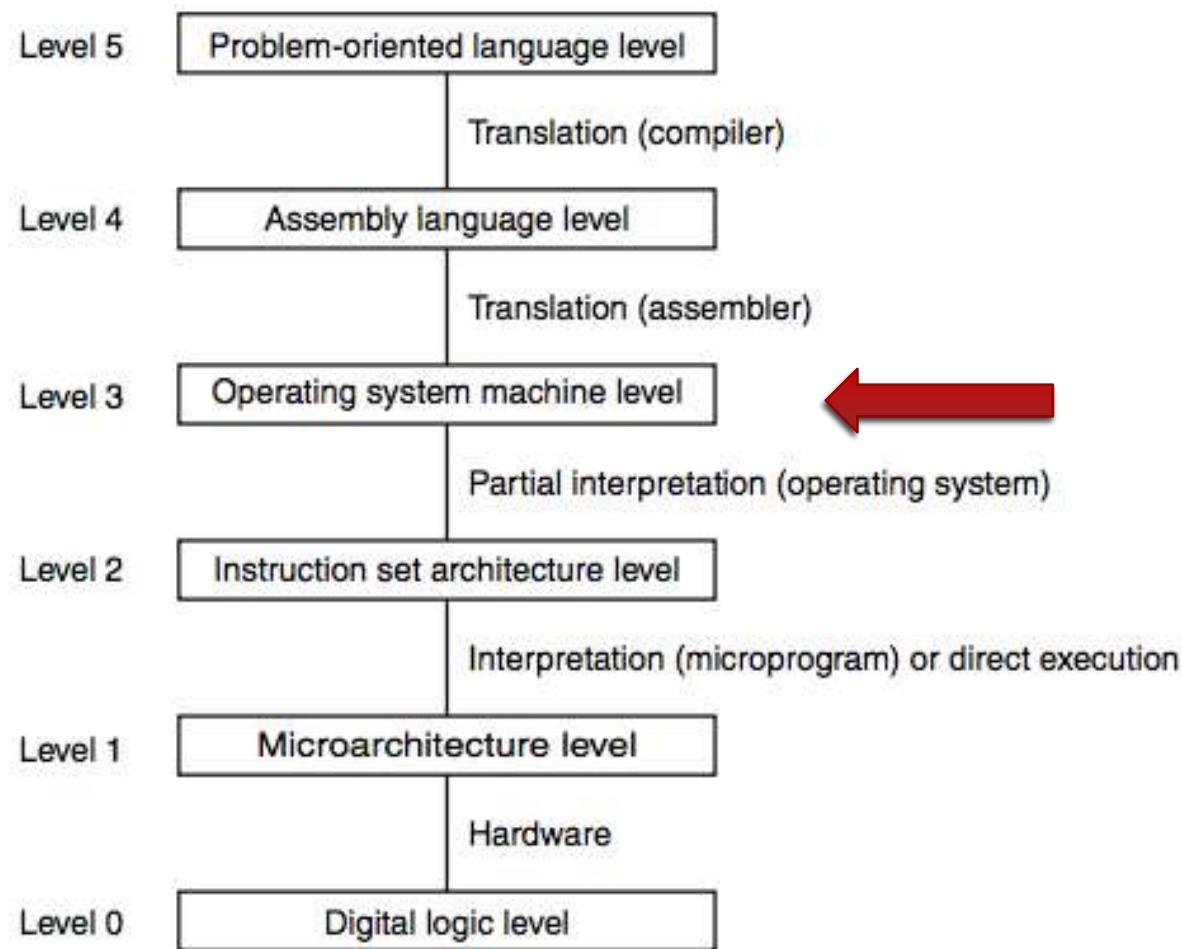


- ↗ The Operating System Machine Layer
- ↗ Program Discontinuities
 - ↗ Interrupts
 - ↗ TRAPs
 - ↗ Exceptions

Operating System Machine Layer

- ↗ Operating Systems are about sharing resources and protecting users from themselves and others.
- ↗ Certain operations require **specialized knowledge** and **protection**:
 - ↗ Specific knowledge of I/O device registers and the sequence of operations needed to use them
 - ↗ I/O and memory resources shared among multiple users/programs; a mistake could affect lots of other users/processes!
- ↗ Not every programmer knows (or wants to know) this level of detail, so we abstract them in an Operating System Machine Layer
- ↗ We transparently provide service to I/O devices and we notify user programs of unexpected situations
- ↗ We provide ***service routines*** or ***system calls*** (as part of operating system) to safely and conveniently perform low-level, privileged operations

Road Map



What are Program Discontinuities

- Interrupts
 - An I/O device is reporting a completion or an error (e.g. “Read completed”)
- TRAPs
 - The program is calling a privileged operating system subroutine (e.g. “Read a line from a file”)
- Exceptions
 - Something unanticipated has happened
 - Hardware error in the CPU or memory
 - Program error (e.g. illegal opcode, divide by zero)

Program Discontinuities

- ↗ We combine the discussion of program discontinuities because they are all handled in very similar ways
 - ↗ Save the state of the CPU
 - ↗ Raise the CPU privilege level
 - ↗ Call an operating system routine
 - ↗ Restore the state of the CPU and the privilege level
 - ↗ Resume the executing program where it left off
- ↗ Does this resemble an unplanned subroutine call?
 - ↗ (The microcode isn't subject to program discontinuities; its job is to provide that abstraction for the machine language program)
- ↗ Properties
 - ↗ Synchronous with the program (TRAPs and some exceptions) or asynchronous (interrupts and other exceptions)
 - ↗ Anticipated by the programmer (TRAPs) or unanticipated (interrupts and exceptions)

Interrupts

- What are they?
- Why do they exist?
- Generation of Interrupt Signal
 - Device
 - Priority
 - FSM Mods

What are they?

- ↗ Modifications to the hardware of the datapath and I/O system and additional software to allow an external device to cause the CPU to stop current execution and execute a "service" routine and then resume execution of the original program. Hence, an **interrupt**.
- ↗ Remember the INT signal that the LC-3 microsequencer can test? That signal is what an interrupt ultimately needs to raise.

Why Do They Exist?

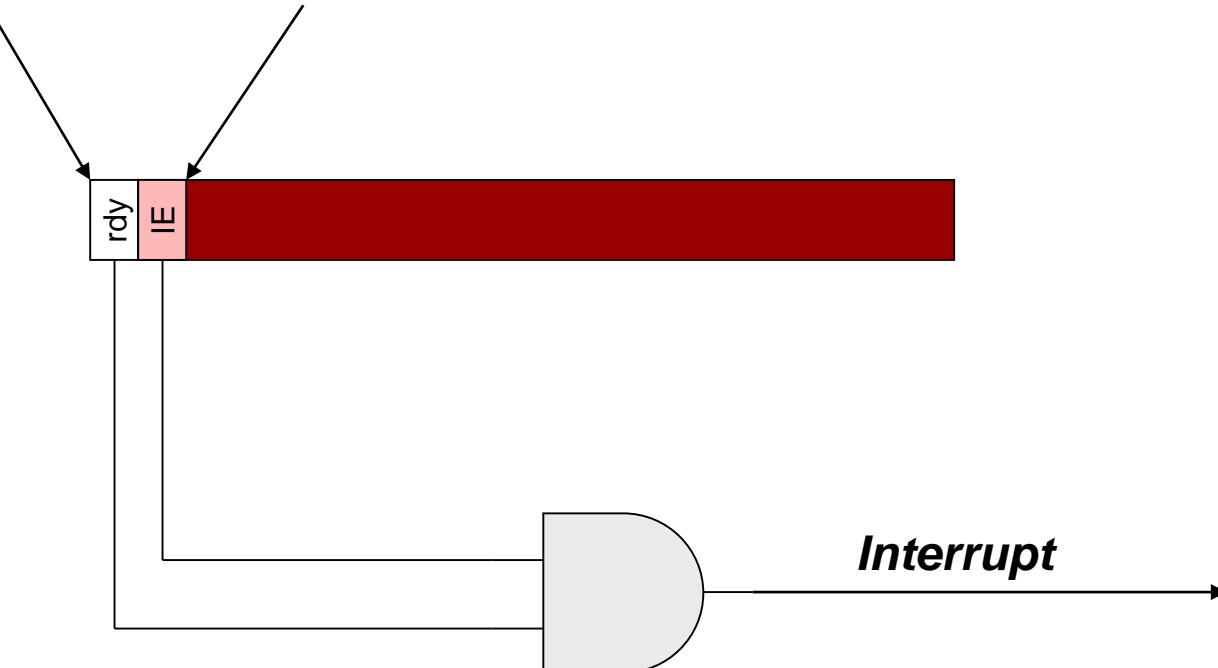
- Interrupts can be significantly more efficient than polling and are especially useful in an environment where there are numerous devices and multiple concurrent activities
- Polling on the other hand is appropriate where there is a high likelihood of quick success or the CPU has nothing better to do

*An interrupt is an **unscripted subroutine call**, triggered by an external event.*

Device Status Register

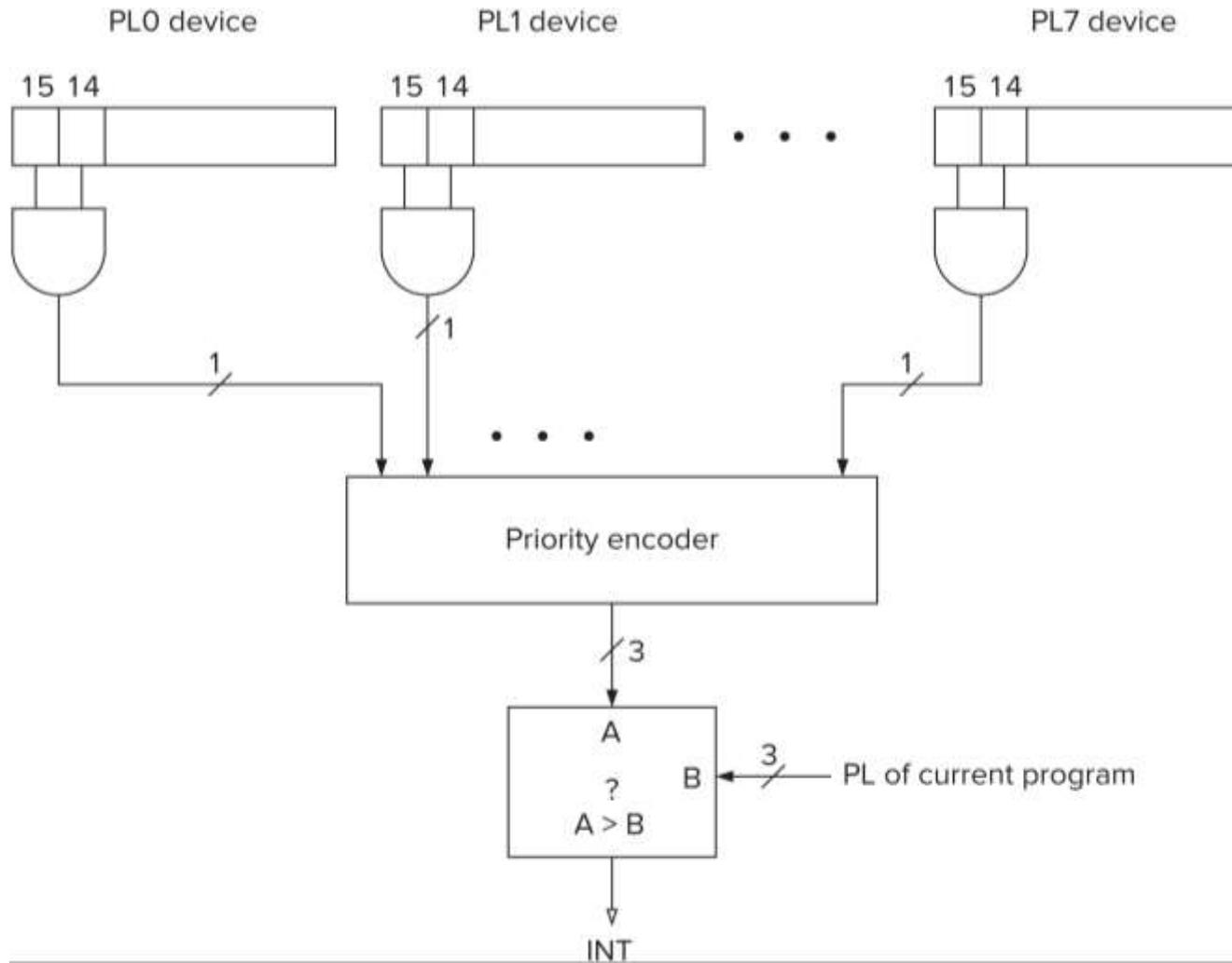
Ready Bit

Interrupt Enable Bit



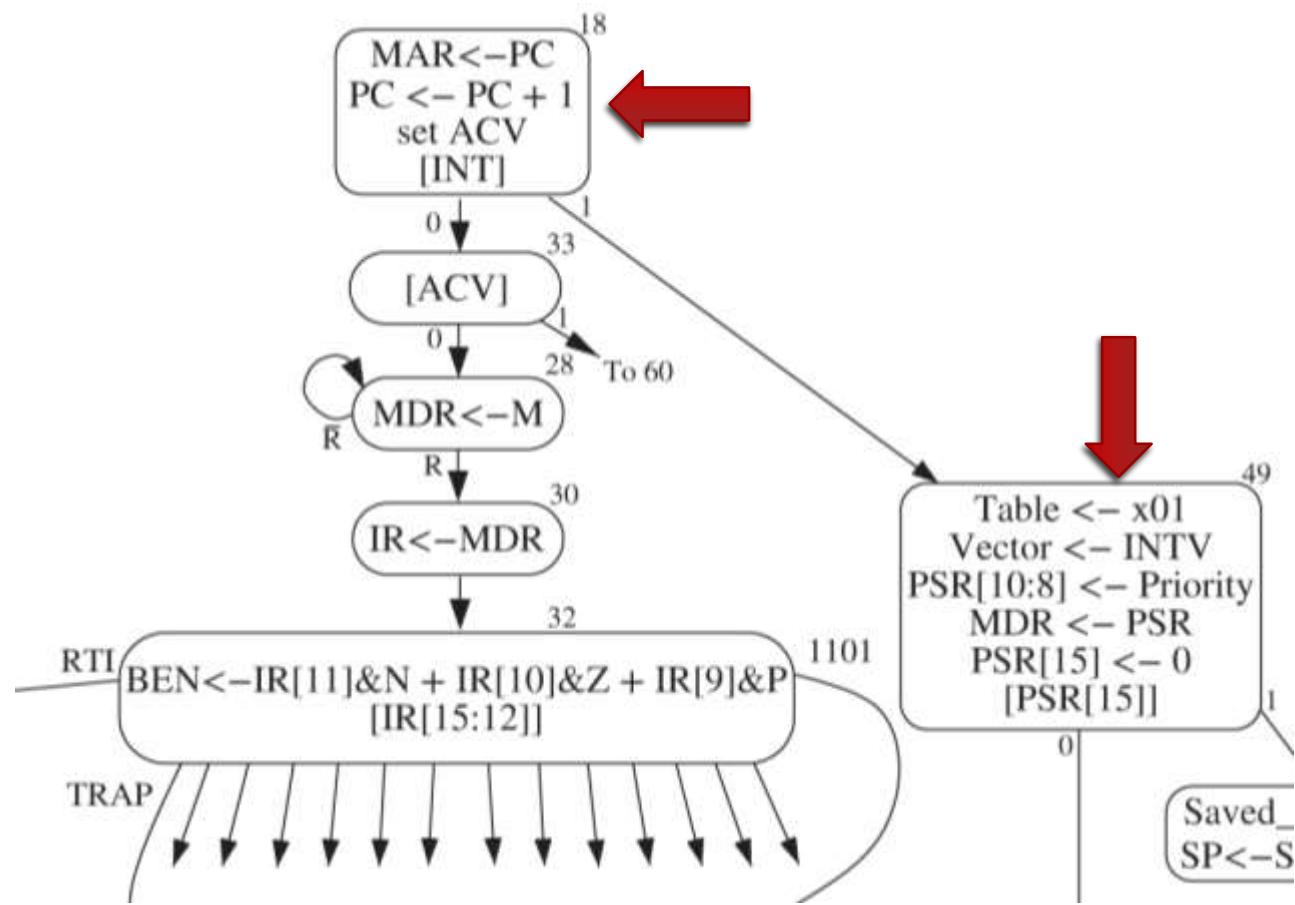
- When a device goes “ready” and its IE bit is set, it will generate an interrupt signal
- There are several more steps that must allow the interrupt signal to pass before the microsequencer can see it

Interrupt Circuitry, for Completeness



- When a device (or devices) interrupts, the priority encoder outputs the highest priority with an interrupt signal
- If the priority is greater than the priority in the PL register, the comparator asserts the INT signal to the microsequencer
- The new priority is stored in PL
- The interrupting device ID is stored in the INTV register (circuit not shown)

Detecting Interrupts



- When the I/O device is allowed to signal an interrupt, the INT control signal to the microcode FSM is asserted
- The first state in the FETCH cycle tests to see if INT is asserted
- If it is, the microcode transfers to state 49 and sets up a call to the interrupt service routine

Where to Save Processor State?

- ↗ Can't use general purpose registers
 - ↗ Programmer doesn't know when interrupt might occur, so she can't prepare by saving critical registers.
 - ↗ When resuming, need to restore state exactly as it was.
- ↗ Can't use memory allocated by service routine
 - ↗ Must save state before invoking routine, so the hardware wouldn't know where.
 - ↗ Also, interrupts may be nested – that is, an interrupt service routine might also get interrupted!
- ↗ **Use the existing supervisor stack!**
 - ↗ Top of the stack is already known in R6.
 - ↗ Push state to save, pop to restore.

Supervisor Stack, for Completeness

- ↗ Interrupts are handled on the supervisor-mode stack to protect them from user mode programs
 - ↗ R6 points to the supervisor-mode stack when the PSR is set to Supervisor mode (bit 15) and to the user-mode stack when it's not
- ↗ We want to use R6 as stack pointer.
 - ↗ So that our PUSH/POP routines still work.
- ↗ When swapping between modes
 - ↗ Supervisor Stack Pointer (SSP) is saved in Saved.SSP.
 - ↗ User Stack Pointer (USP) is saved in Saved.USP.
- ↗ An interrupt in User mode will switch to Supervisor mode, save R6 to Saved.USP, and load R6 from Saved.SSP.

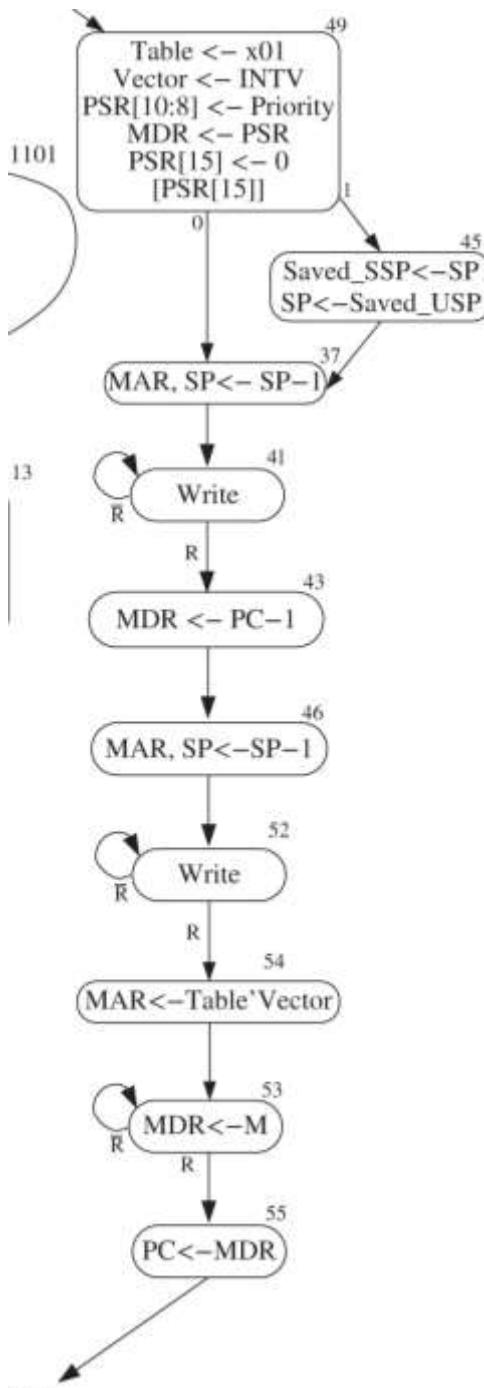
What is the Processor State?

- ↗ What three items are needed to completely capture the state of a running process on the LC-3?
- ↗ **(1) Processor Status Register**
 - ↗ Privilege [15], Priority Level [10:8], Condition Codes [2:0]
 - ↗ This register doesn't really exist, but the hardware pushes the values on the stack using the following format



- ↗ **(2) Program Counter**
 - ↗ Pointer to next instruction to be executed.
- ↗ **(3) General Registers**
 - ↗ All temporary state of the process that's not stored in memory.
- ↗ Privilege (P) has only two values, 0 and 1, but a lot of synonyms.
 - ↗ 1 is user mode or unprivileged mode
 - ↗ 0 is system, supervisor, kernel, or privileged mode

Handling an Interrupt



1. Set Table to x01, Vector to INTV (i.e., interrupting device ID)
2. If **PSR[15] == 1** (user state),
Saved.USP = R6, R6 = Saved.SSP.
3. Push PSR and PC-1 to Supervisor Stack.
4. Set **PSR[15] = 0** (supervisor mode).
5. Set **PSR[10:8] = priority of interrupt being serviced**.
6. Set MAR = Table'Vector, where **Vector** = 8-bit interrupt vector provided by interrupting device (e.g., keyboard = x80).
7. Load memory location (M[Table'Vector]) into MDR.
8. Set **PC = MDR**; now first instruction of ISR will be fetched next.
9. Go back to the first state of FETCH (state 18)

Returning from Interrupt

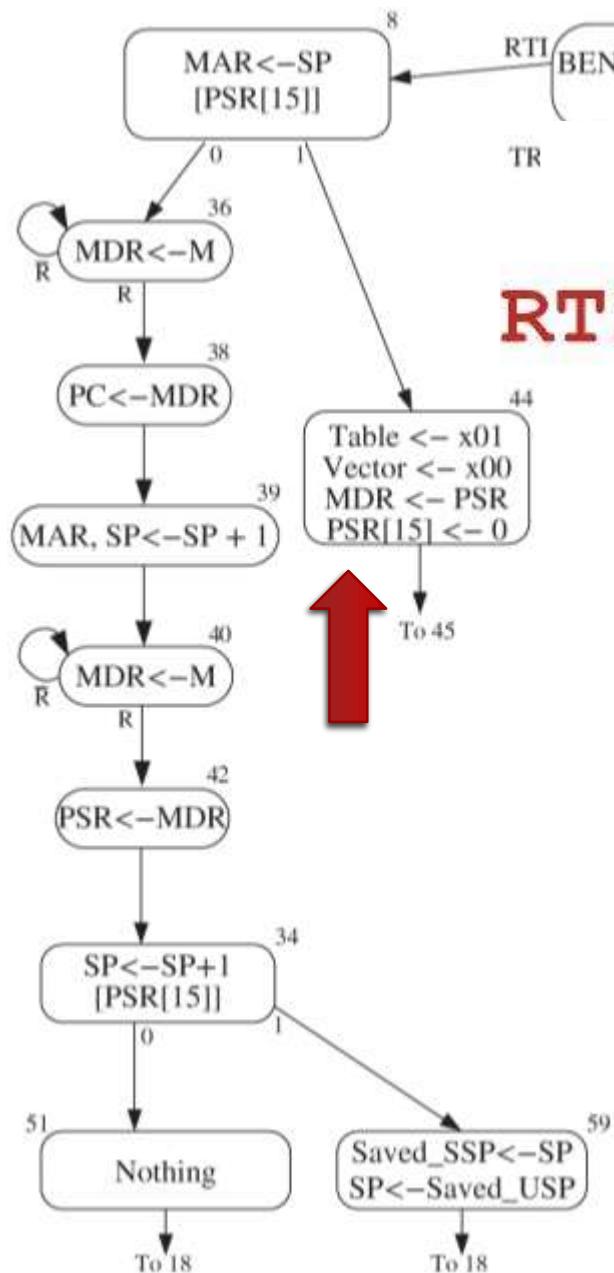
- Special instruction – RTI – that restores state.

RTI	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

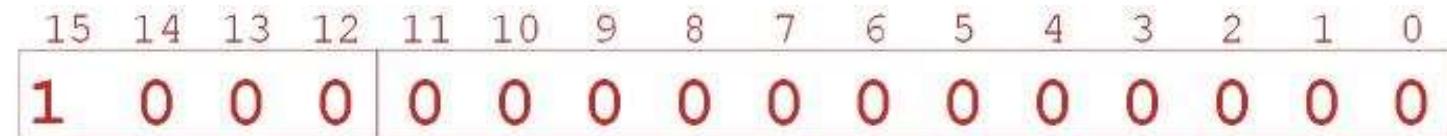
1. Pop PC from supervisor stack. ($PC = M[R6]$; $R6 = R6 + 1$)
2. Pop PSR from supervisor stack. ($PSR = M[R6]$; $R6 = R6 + 1$)
3. If $PSR[15] = 1$, $\text{Saved.SSP} = R6$, $R6 = \text{Saved.USP}$.
(If going back to user mode, need to restore User Stack Pointer.)

- RTI is a privileged instruction.
 - Can only be executed in Supervisor Mode.
 - If executed in User Mode, causes an exception.
(More about that later.)

Returning from an Interrupt (RTI)



↗ Special instruction – RTI – that restores state.

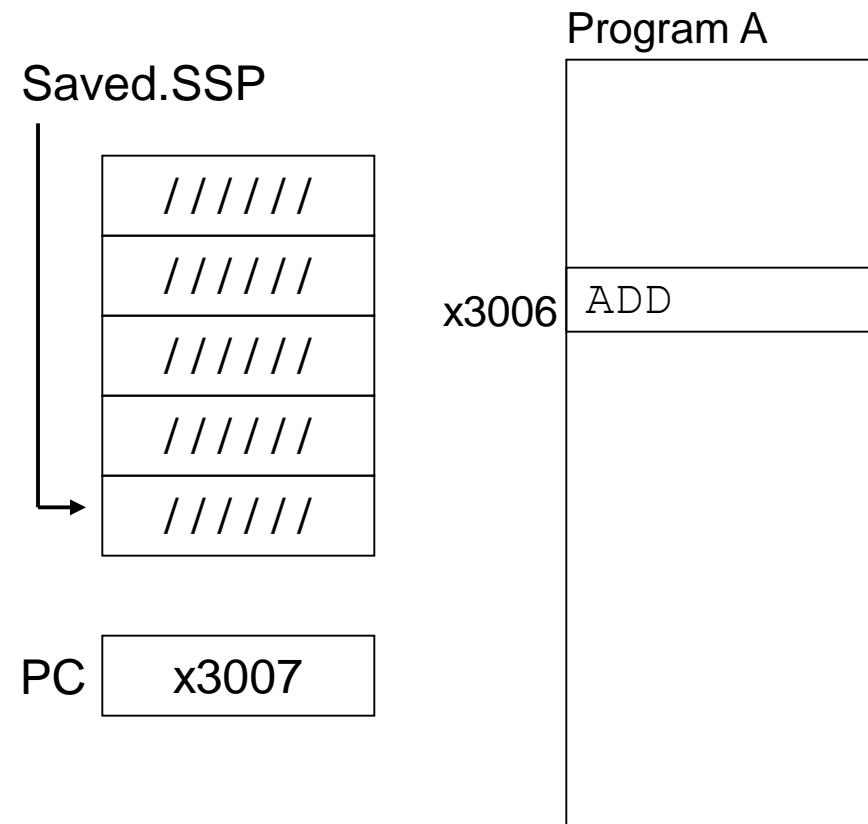


1. Pop PC from supervisor stack. ($PC = M[R6]$; $R6 = R6 + 1$)
2. Pop PSR from supervisor stack. ($PSR = M[R6]$; $R6 = R6 + 1$)
3. If $PSR[15] = 1$, $\text{Saved.SSP} = R6$, $R6 = \text{Saved.USP}$.
(i.e., if going back to user mode, restore User Stack Pointer.)

↗ RTI is a privileged instruction.

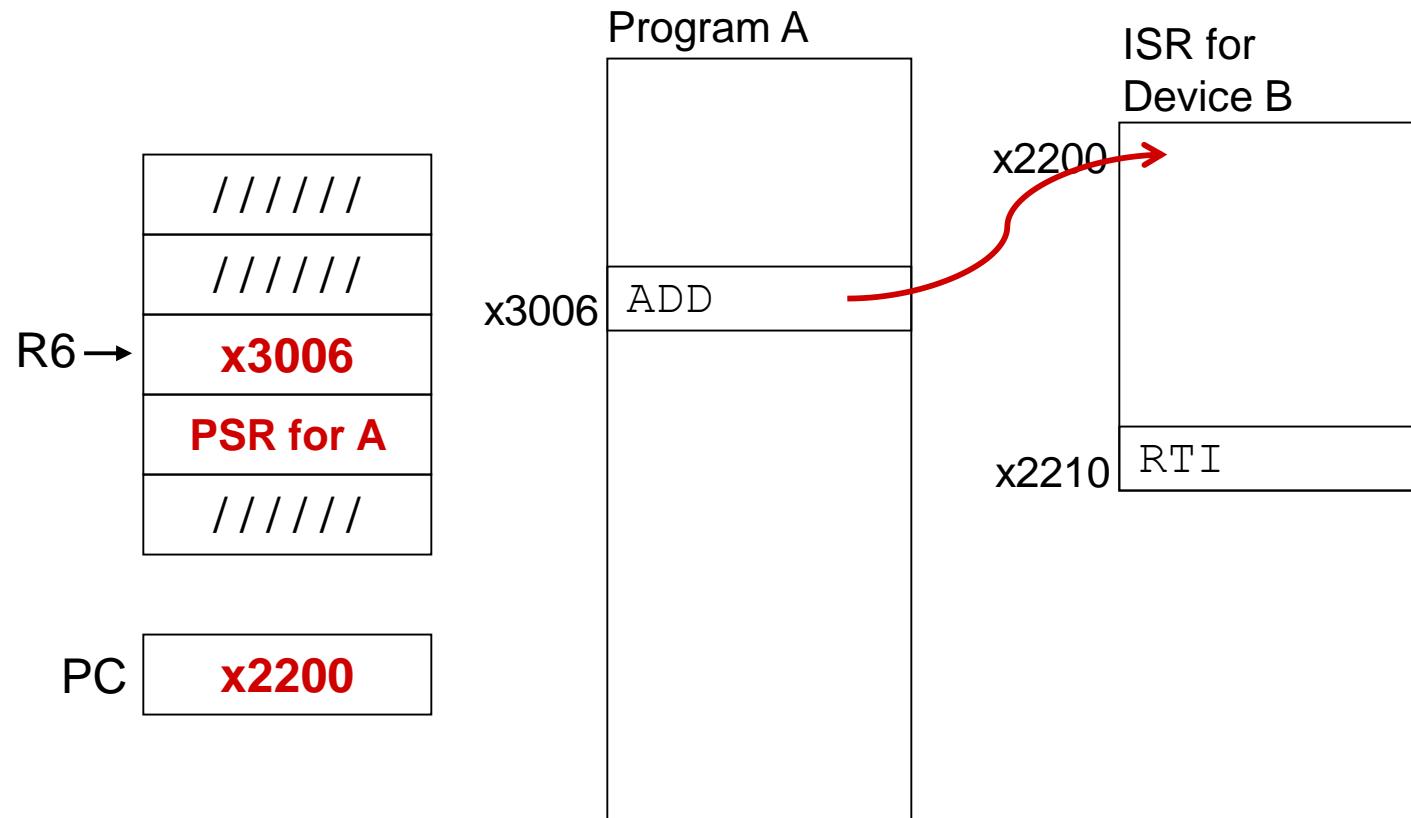
- ↗ Can only be executed in Supervisor Mode.
- ↗ If executed in User Mode, causes an exception.
(More about that later.)

Example (1)



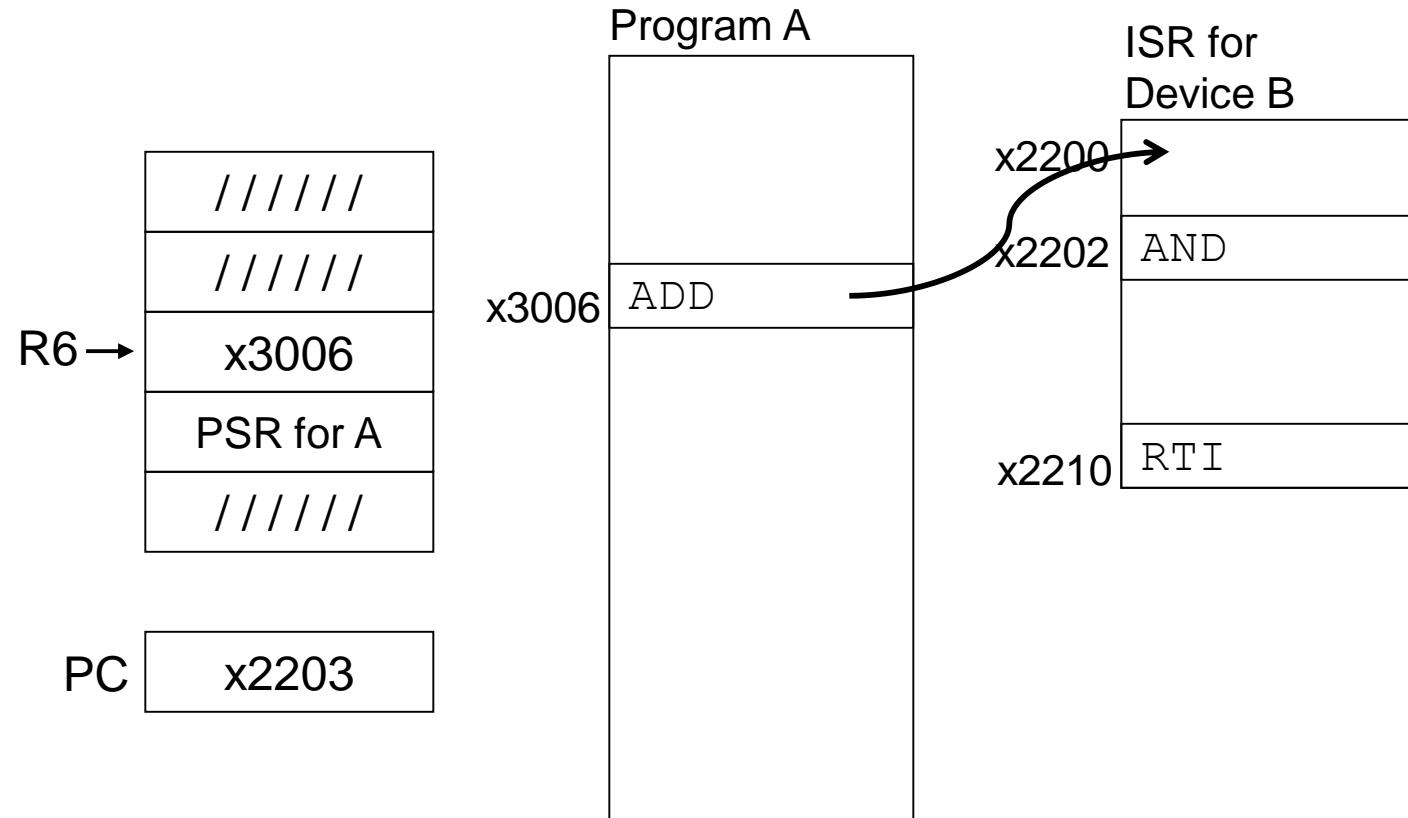
Fetching ADD at location x3006 when Device B interrupts; PC already incremented.

Example (2)



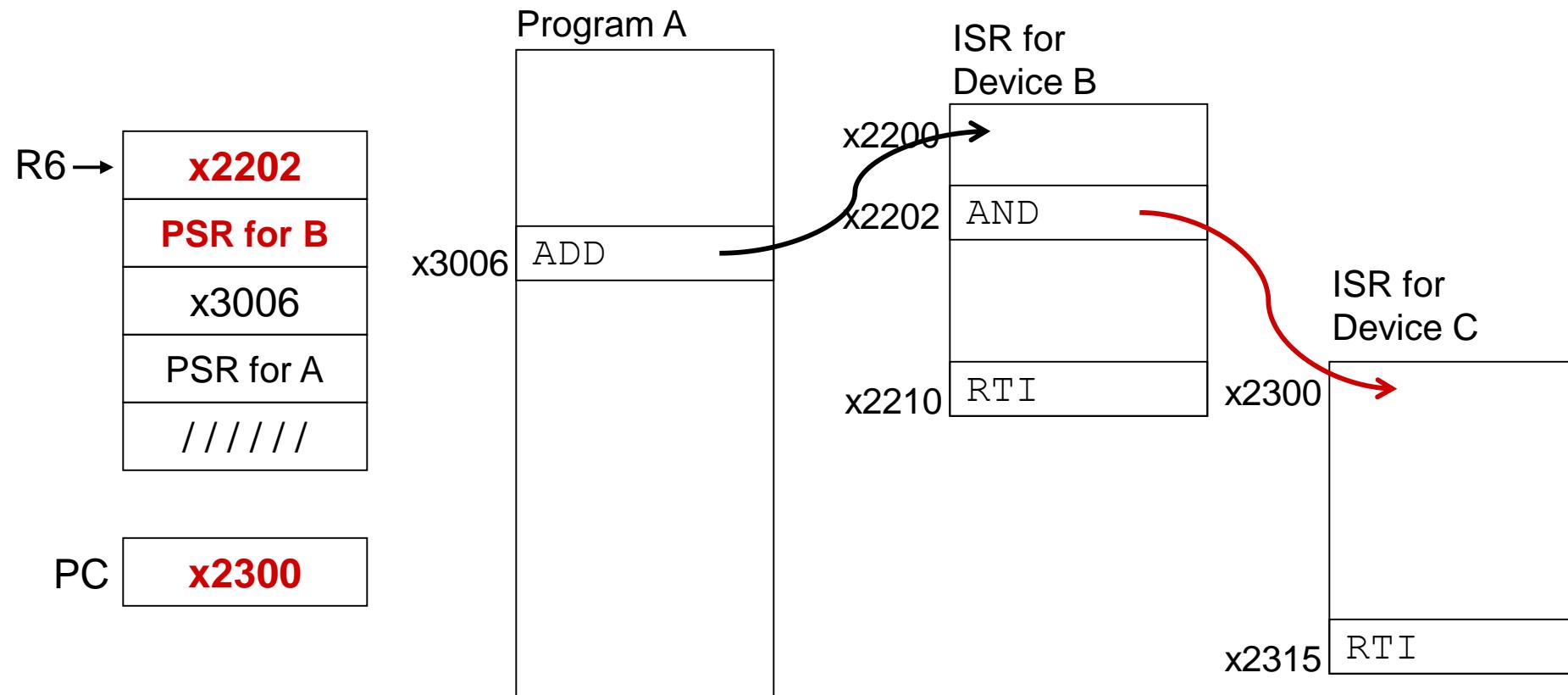
Saved.USP = R6. R6 = Saved.SSP.
Push PSR and PC onto stack, then transfer to
Device B service routine (at x2200).

Example (3)



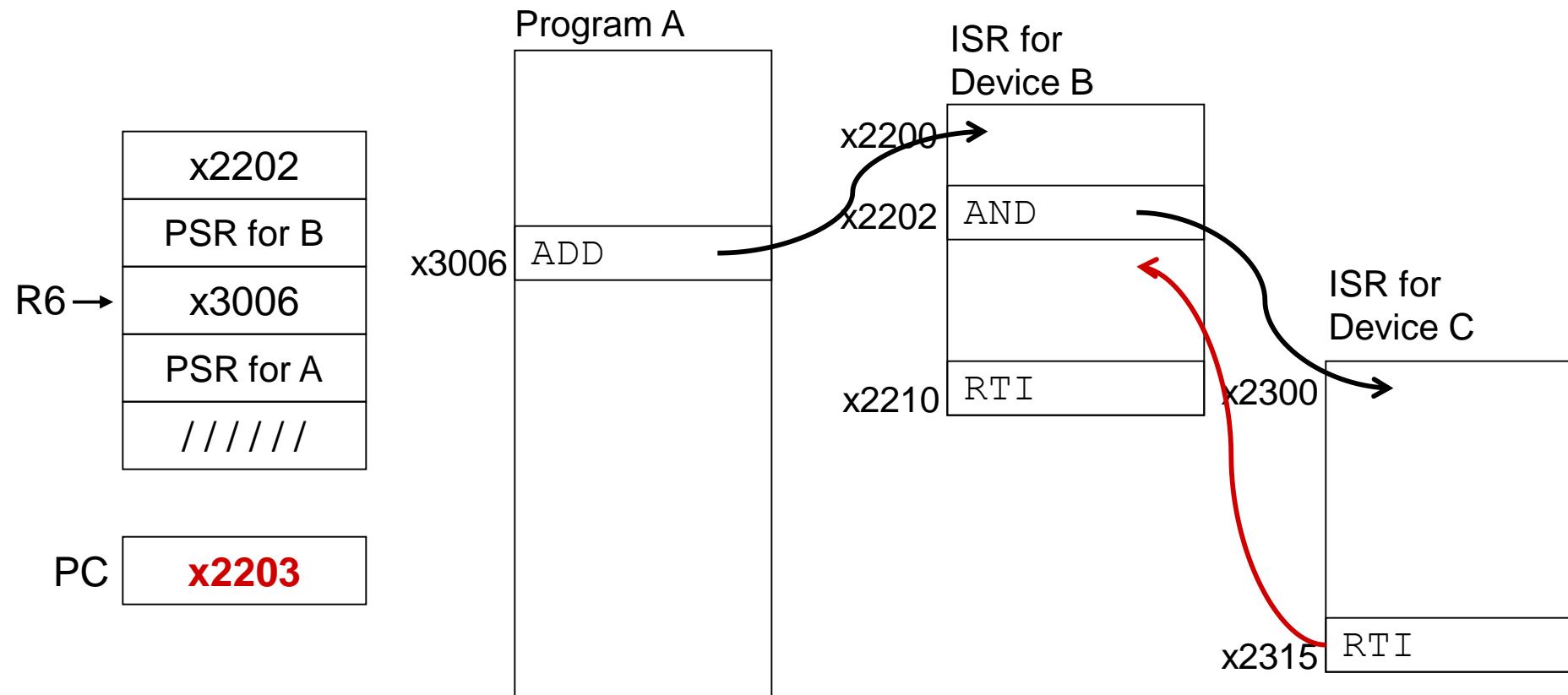
Executing AND at x2202 when Device C interrupts.

Example (4)



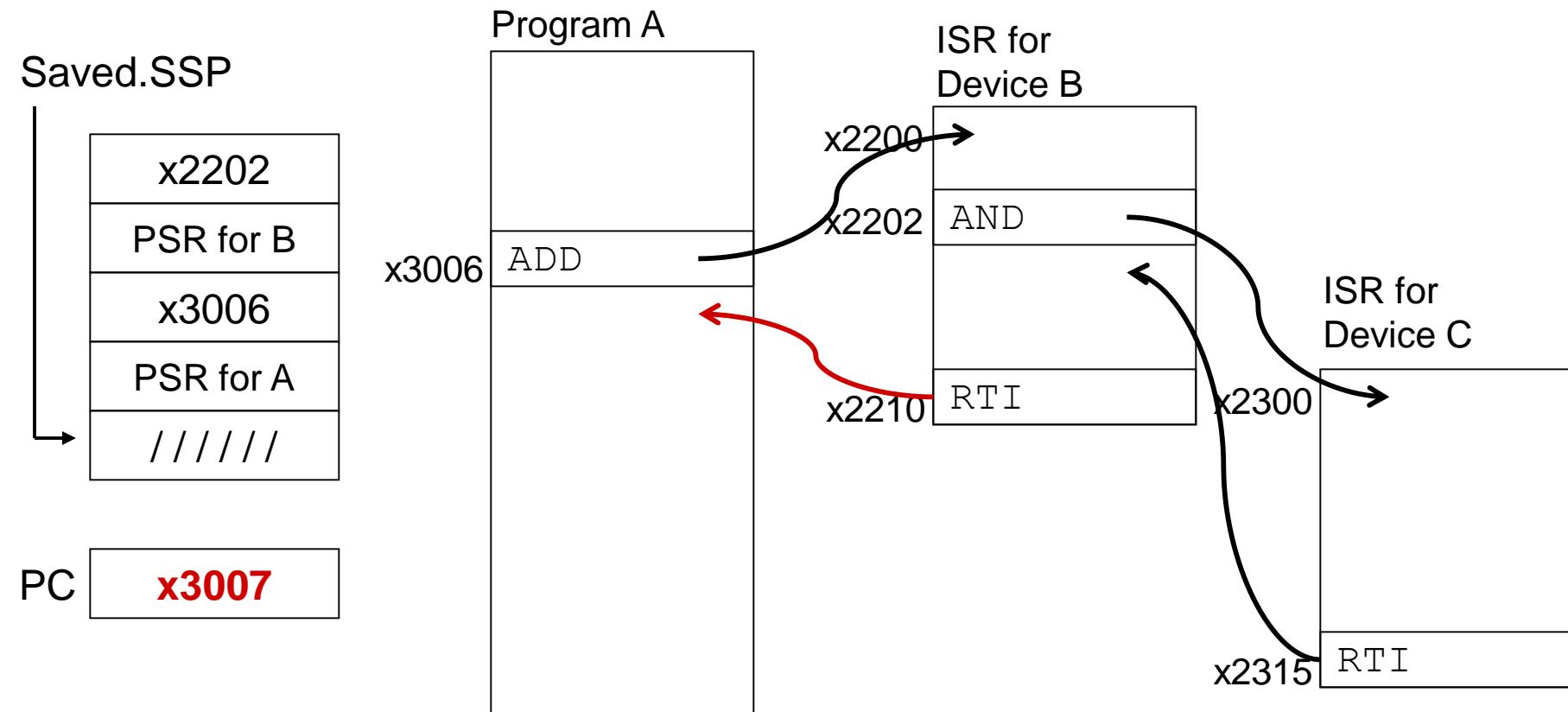
Push PSR and PC onto stack, then transfer to
Device C service routine (at x2300).

Example (5)



Execute RTI at x2315; pop PC and PSR from stack.

Example (6)



Execute RTI at x2210; pop PSR and PC from stack.
Restore R6. Continue Program A as if nothing happened.

Operating Systems?

- ↗ Most of this we'll save for CS 2200, but...
- ↗ Just how are we going to ask the Operating System to do things for us?
- ↗ Turns out that something like a subroutine call would work, but...
- ↗ We don't know where the operating system will be stored in memory (or at least we don't know where the service procedure we want is stored) and we only have user-mode privileges
- ↗ So how can we call it?
- ↗ We'll use a special instruction, TRAP
 - ↗ It will jump indirectly through a table provided by the operating system
 - ↗ It will elevate our privileges to supervisor mode

TRAP vs JSR(R)

↗ TRAP

- ↗ Uses trap vector table
 - ↗ (Can call from anywhere)
 - ↗ (TV table is loaded by the OS)
- ↗ Normally calls system functions
 - ↗ I/O, resource sharing, etc.
- ↗ Written very carefully!
- ↗ If in user state, switch to supervisor state to allow privileged action

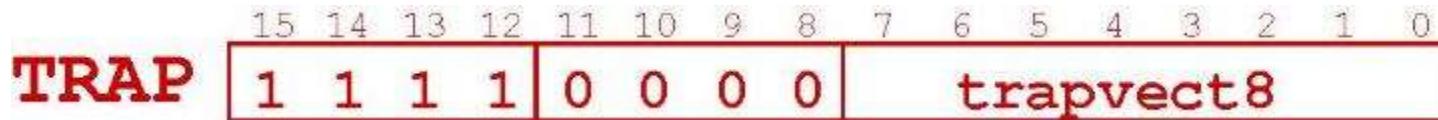
↗ JSR(R)

- ↗ Nearby (JSR)
- ↗ Anywhere (JSRR)
 - ↗ with some work
- ↗ Routine abstraction
- ↗ Code reuse/libraries
- ↗ No protection mechanism

LC-3 TRAP Mechanism

- ↗ *1. A set of service routines.*
 - ↗ They are part of operating system – service routines start at arbitrary addresses within the OS
(convention is that system code lives between x0200 and x3000)
 - ↗ Supports up to 256 service routines
- ↗ *2. Using a table of starting addresses.*
 - ↗ Stored at **x0000** through **x0OFF** in memory
 - ↗ Called **System Control Block** in some architectures
 - ↗ Initialized by the operating system
- ↗ *3. TRAP instruction.*
 - ↗ Used by program to transfer control to an operating system routine
 - ↗ 8-bit trap vector names one of the 256 service routines
 - ↗ Saves PSR and PC on via the R6 stack and gains privilege just like Interrupts

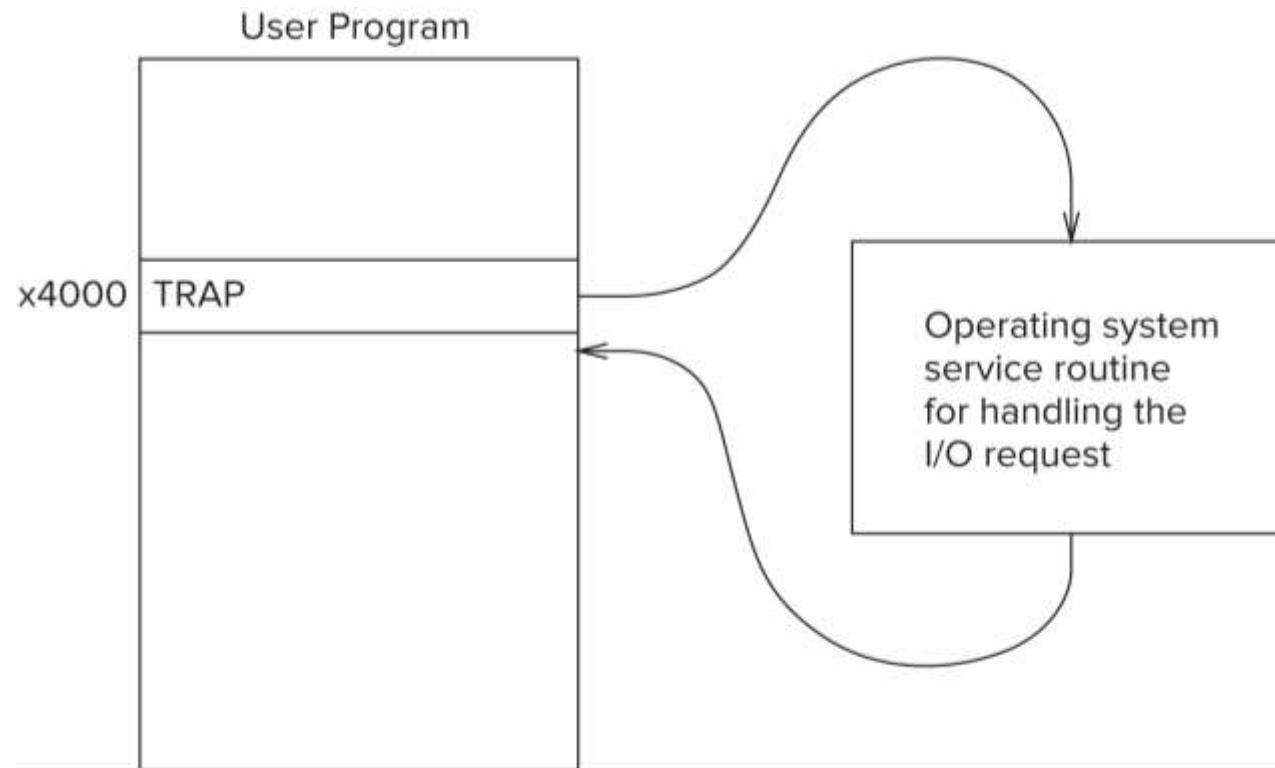
TRAP Instruction



➤ Trap vector

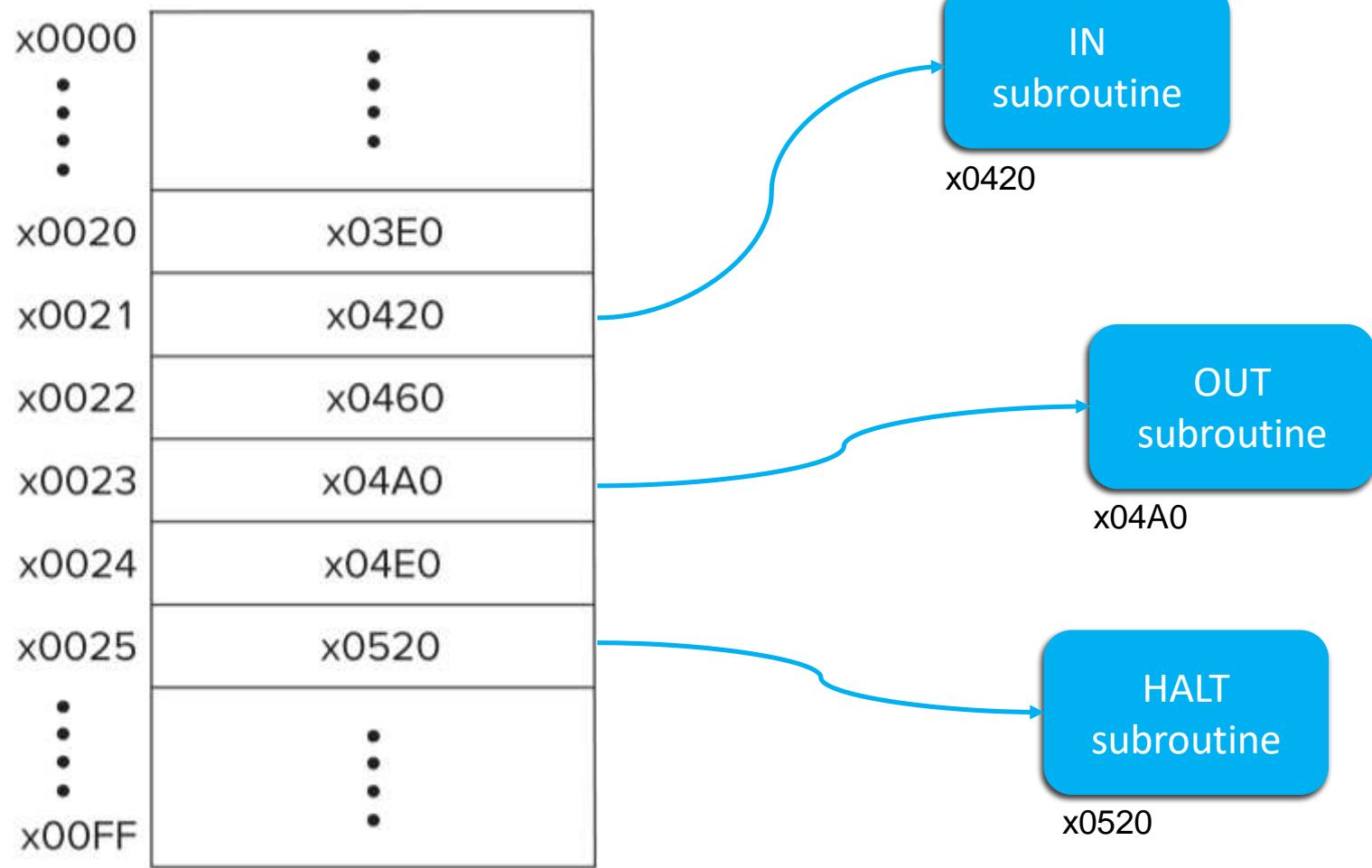
- identifies the address of the system call to invoke
- 8-bit index into Trap Vector Table
 - in LC-3, this table is stored in memory at **0x0000 – 0x00FF**
 - 8-bit trap vector is zero-extended into 16-bit memory address

LC-3 Trap



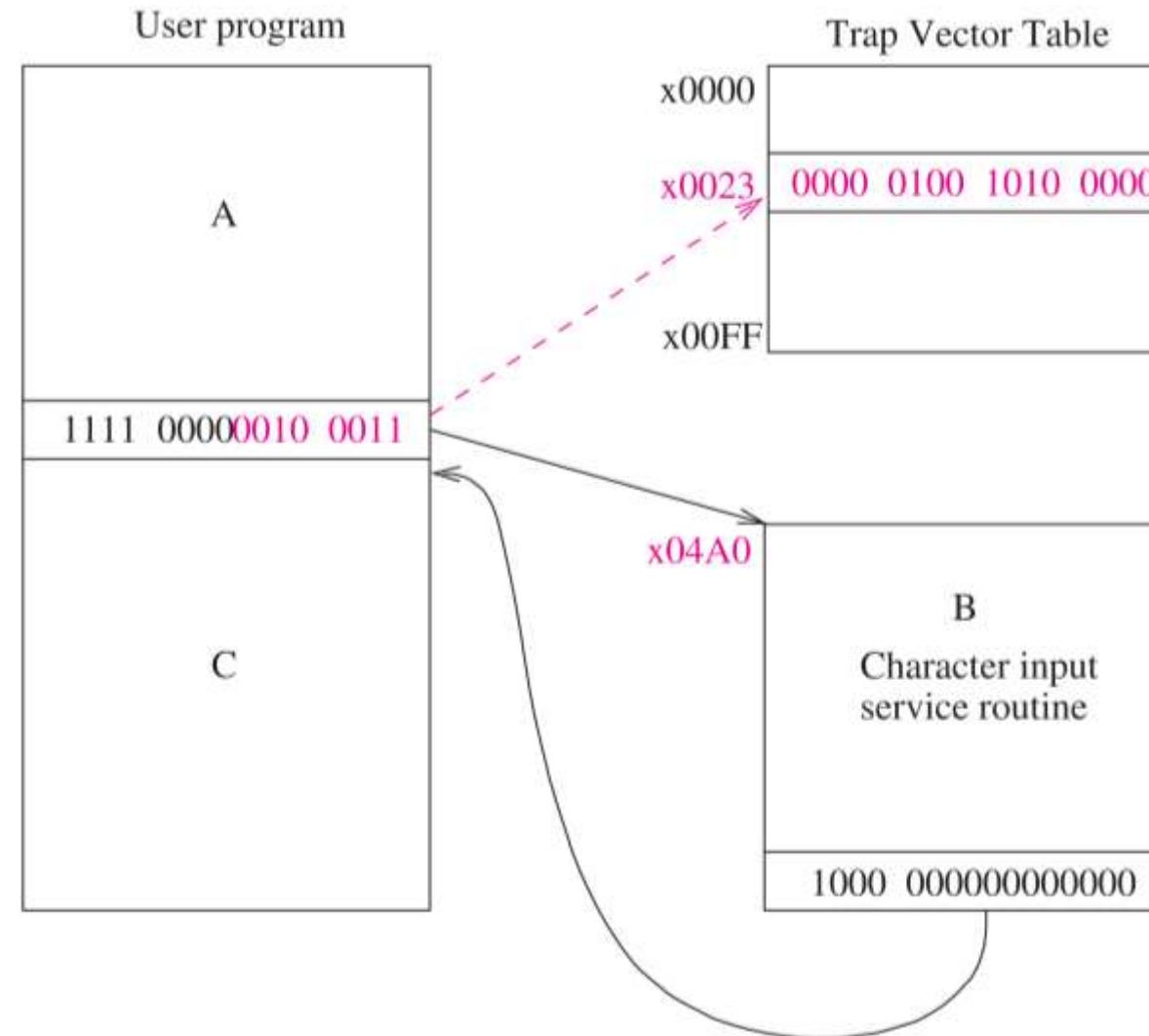
- That's a lot of words! What do they mean!
- Think of TRAP as a special *indirect* JSR with a choice of memory location from 0 through 255 (i.e. let's call that *trapvect*).
- The TRAP instruction saves state just like an Interrupt on the system stack

The Trap Vector Table



Trap Vector Table

Calling the OUT Subroutine with TRAP



Common LC-3 Trap Numbers

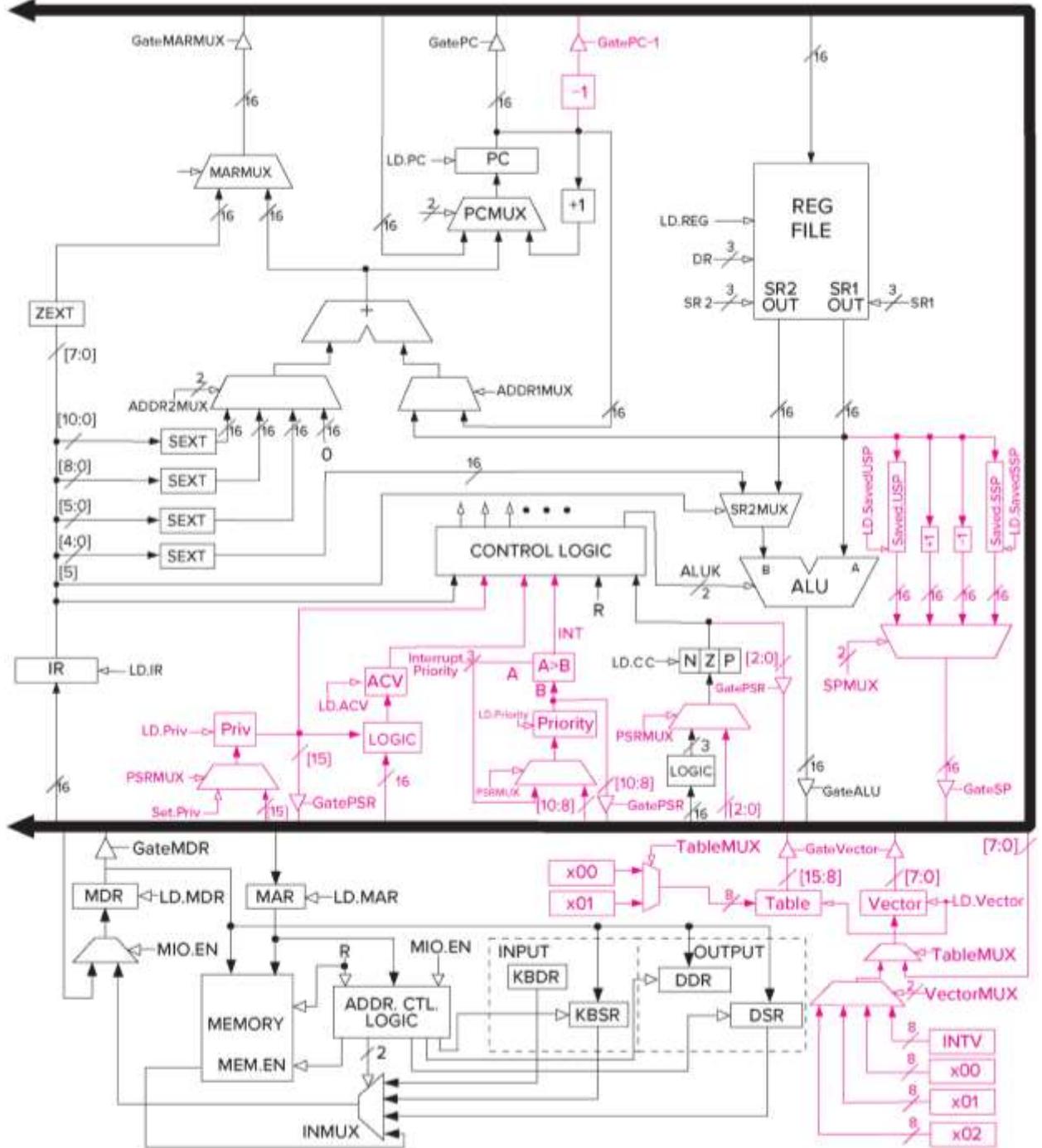
TRAP



<i>Vector</i>	<i>Routine</i>
x23	IN – prompt and read a character from the keyboard (R0)
x21	OUT - output a character to the monitor (R0)
x25	HALT - halt the program
x20	GETC - read a character from the keyboard (R0)
x22	PUTS - output a string, 1 char per word ending with x0000, address in R0
x24	PUTSP - output a string, 2 characters per word ending with a word of x0000, address in R0

- ↗ Now that we understand Interrupts and TRAPs, Exceptions are easy
- ↗ Synchronous exceptions in the LC-3
 - ↗ Divide by zero (no divide in LC-3)
 - ↗ Illegal instruction (opcode=D, Table'Vector is 0x0101)
 - ↗ Privileged instruction (RTI in user mode, Table'Vector is 0x0100)
 - ↗ Address Violation (ACV, Table'Vector is 0x0102)
- ↗ Asynchronous exceptions (not implemented on LC-3)
 - ↗ Processor error
 - ↗ Memory error
- ↗ Handled like Interrupts, but of course the vectors point to exception service routines in the operating system

Program Discontinuity Logic, for Completeness (Fig C.8)



Questions?

Intro to C



So Where Are We Now?

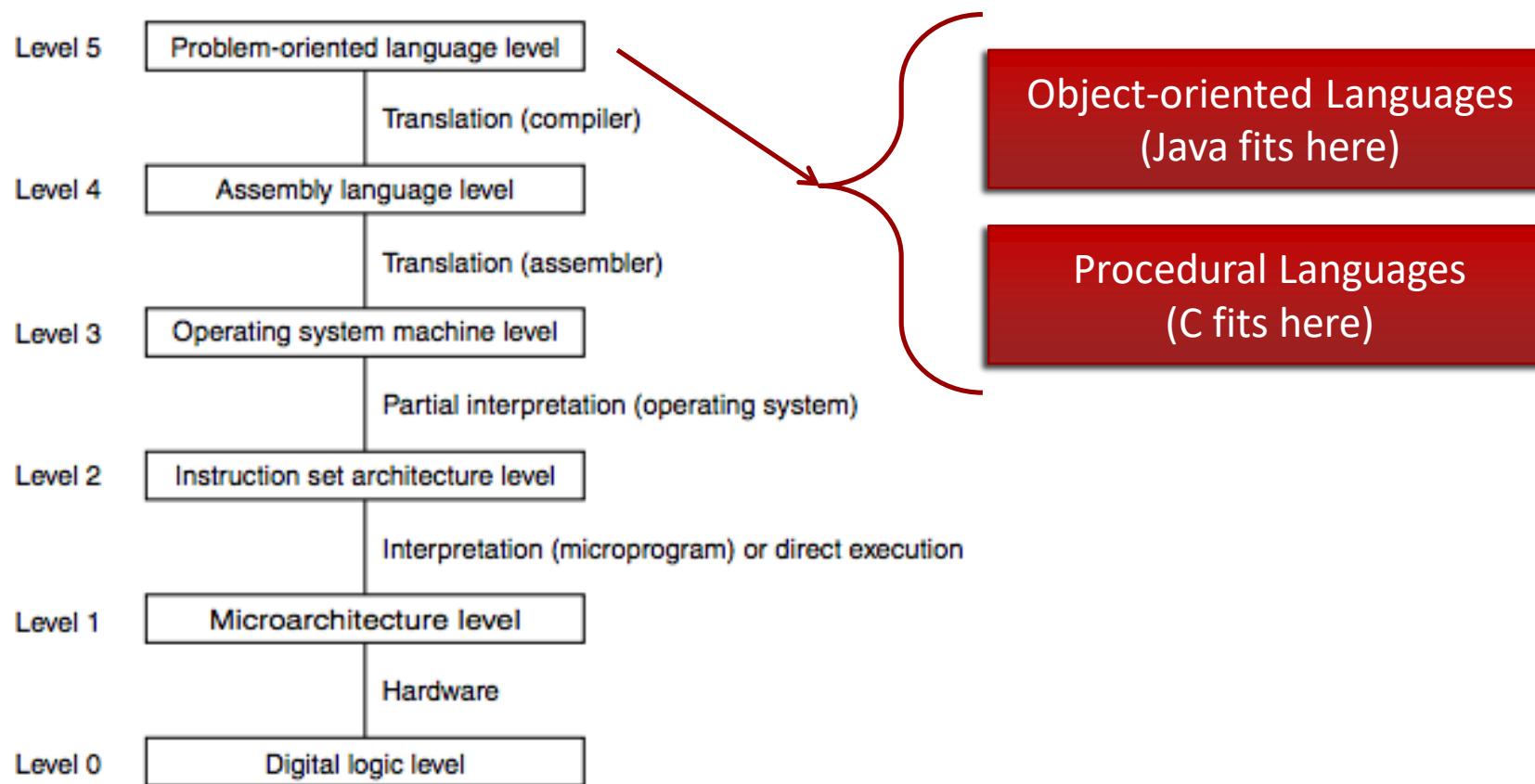
- ↗ We've taken apart a computer from the circuit level to the assembly-language level (and it's not even mid-term)!
- ↗ We know how to work those pieces
- ↗ We're ready to talk seriously about problem-oriented languages

Problem-Oriented Language

- ↗ A language whose statements resemble terminology of the user application-oriented language rather than machine language.
- ↗ In other words a language in which algorithms are expressed in human terminology (math, science, business) as opposed to machine implementation terms.
- ↗ There have been and are **lots** of them.

One More Abstraction...

- These days, we even have abstraction layers within the problem-oriented language layer



- ↗ Give Symbolic Names to Values
- ↗ Provide Expressiveness
- ↗ Abstract the Underlying Hardware
- ↗ Enhance Code Readability
- ↗ Provide Safeguards Against Bugs

- ↗ Ancestors
 - ↗ Fortran
 - ↗ Algol 60
 - ↗ PL/I
- ↗ Parents
 - ↗ (CPL, BCPL, B) -> C
- ↗ Descendents
 - ↗ Perl
 - ↗ Java
 - ↗ Python
 - ↗ C++
 - ↗ C#

The C Language

- ↗ BCPL (Typeless) - 1967 - Operating Systems
- ↗ B - Ken Thompson - First version of Unix
- ↗ C - 1972 - Dennis Ritchie - Implemented on PDP-11
- ↗ Unix rewritten in C in 1974
- ↗ Publication in 1978 of "The C Programming Language"
by Brian Kernighan and Dennis Ritchie
- ↗ ...
- ↗ ANSI C (1989)
 - ↗ Major updates to C standard in 1994 and 1999
 - ↗ We will use C99 in CS 2110 (GNU C compiler)

Learning to Program



CS1



Learning to Program

CS2



So We're Ready for a Fast, Modern Jet, Right?



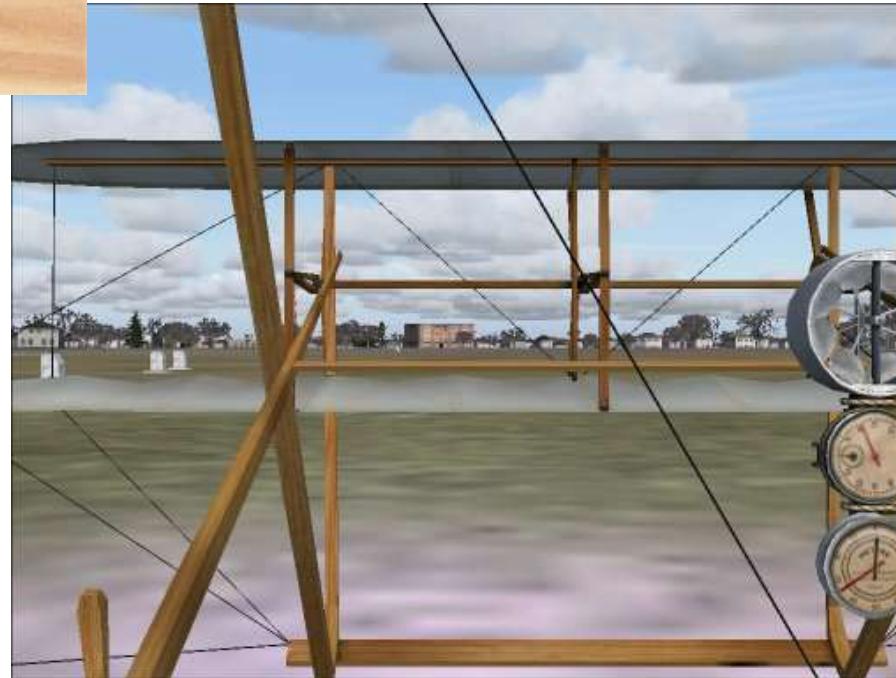
CS2110



Well, C is More Like...



CS2110



- ↗ C is the language that will allow you to do anything.
- ↗ It is used to write operating systems, other languages, low level hardware drivers, cryptography, networking, etc.
- ↗ It was designed to be easily compiled and to produce compact, efficient code
- ↗ It won't check for many runtime errors
- ↗ Remember: It was invented to implement system software for a **small** computer

- ↗ C is not your friend
- ↗ C trusts you and will do exactly what you ask: It is quite certain you are a careful, knowledgeable programmer. Don't disappoint it!
- ↗ All the symbols mean something!
- ↗ "I'll just type something in and the compiler will fix it for me" just won't work
- ↗ Read and reread the C book! And there are plenty more tutorial resources on the net

- Many problem-oriented programming languages have features that try to prevent the programmer from doing something wrong or letting you know when you do, e.g.
 - Java does not allow you access memory by address
 - Java throws an exception when you have an array out of bounds
 - C is not one of these languages. Why?

Differences Between Java and C

- ↗ C is about 45 years old and pre-dates Java by about 25 years
- ↗ C is procedural; no objects or classes
- ↗ C structs are used in place of classes
- ↗ Pointers are used in place of object references
- ↗ No overloading of function names; each function must have a unique name
- ↗ C does not include strings
 - ↗ Strings are arrays of characters
 - ↗ And characters are 8-bit integers
- ↗ C does not have print or I/O built-in
 - ↗ For strings and printing, use standard C libraries

Expressions and Control Statements

- Expressions are very similar
 - All expressions (including assignments) yield a value.
 - Order of precedence is the same
 - No “new”, “instanceof”, or >>> (from Java)
 - C does have: “sizeof”, unary &, unary * and binary ->
- Control statements are quite similar
 - Old friends “if”, “while”, “do-while”, “for”, “switch”, “break”, “continue”, “return”

Getting Ready for HW 7

- ↗ We're going to introduce several topics that you'll need to complete Homework 7.
- ↗ You are going to take the terms and examples we present and look them up in K&R, Patt, and/or the internet to get clarification if you need it.
- ↗ Then, after Homework 7 is started, we'll come back and look at some of these topics in more detail.

What We Need for HW 7

- ↗ We'll do these topics first:
 - A. Data types
 - B. Strings
 - C. printf()
 - D. C preprocessor (#define, #include)
 - E. Structs
 - F. Pointers
 - G. Functions
 - H. Command Line Arguments (argc, argv)

- Integer types
 - [unsigned] char 8 bits
 - [unsigned] short [int] 16 bits
 - [unsigned] int 16/32 bits
 - [unsigned] long [int] 32/64 bits
 - [unsigned] long long [int] 64 bits
- Floating point types
 - float 32 bits
 - double 64 bits
 - long double 80/128 bits
- Aggregate types
 - array
 - struct
 - union (later)
- Pointers (a special kind of integer)

How Big? It Depends.

- ↗ It depends on your platform
 - ↗ char – exactly 8 bits
 - ↗ short int – at least 16 bits
 - ↗ int – at least 16 bits
 - ↗ long int – at least 32 bits
- ↗ You can tell with **sizeof**
- ↗ **sizeof** is a **compile-time** constant reflecting the number of bytes held by a data type **or** instance
- ↗ `sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)`
- ↗ **sizeof** char is 1.

- C historically did not have a Boolean data type.
 - Instead, we can use integers to represent Boolean values.
- Any integer that is 0 evaluates as False
- Any non-zero integer evaluates as True
- Also applies for a char
 - Remember a char is just an 8-bit int in C
 - So the NUL character '\0' evaluates to False
- Also applies for a pointer
 - A memory address that is 0x0 evaluates as False
 - More on this later!

- Strings are arrays of characters
- Strings end with an ASCII NUL (a.k.a. 0 or '\0')
- You can define them like this
 - char mystr[6];
- You can initialize them several ways
 - char mystr[6] = { 'H', 'e', 'l', 'l', 'o', '\0' };
 - char mystr[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
 - char mystr[] = "Hello";
- The latter two are **special cases** in which the C compiler determines the length of the array from the initializer
- There are a number of library functions for dealing with strings, including `strlen()`, `strcpy()`, and `strup()`

- `char s[6] = "hello";`
- Same as:
 - `s .stringz "hello" ;` 6 memory locations, ends with 0
- `s` is the memory address where the string starts
- **Never** use `sizeof(s)` when you want the string length!
 - That is the size of the array (or the size of a pointer), not the actual length of the string.
 - Use `strlen(s)` instead
 - Must `#include <string.h>` first

Question

In C, how many bits are the types *short*, *int*, and *long*?

A. 16, 32, 64

B. 32, 64, 64

C. 32, 32, 64

D. Any of the above



Question

In C, a boolean value occupies

- A. 1 bit
- B. 8 bits
- C. 16 bits
- D. 32 bits
- E. Any size an integer can be.



Question

In C, how do you declare and initialize an array to contain the string
The quick brown fox?

- A. char s[20] = “The quick brown fox”;
- B. char s[20] = {'T', 'h', 'e', ' ', 'q', 'u', 'i', 'c', 'k', ' ',
 'b', 'r', 'o', 'w', 'n', ' ', 'f', 'o', 'x', '\0'};
- C. char s[] = “The quick brown fox”;
- D. Any of the above



- ↗ printf() is a function in the Standard IO Library
 - ↗ (*so you need to #include <stdio.h>*)
- ↗ The first argument is a format string
- ↗ Characters in the format string are copied to standard output (which is typically connected to your screen)
- ↗ Format codes beginning with % reference arguments that come after the format string, in order
- ↗ If you want to end a line, you need to output a newline character ('\n')

Printf() Format Codes

- Some useful format codes are

%d	Decimal integer (int)
%x	Hex integer (int)
%f	Floating number (float)
%s	String (char * or char [])
%c	Character (char)
%p	Pointer (for debugging)

- For example

might print `printf("Person: %s GPA: %f\n", name, gpa);`

Person: Dan GPA: 2.5

The C Preprocessor

- The C preprocessor does two main things
 - File inclusion (#include)
 - Macro expansion (#define)
- Preprocessor directives always start with #

- Conventionally, we only *include* files that end in “.h”
- These consist of declarations (including function prototypes) and macro definitions but no executable code
- If you surround the file name with
 - double quotes, the preprocessor looks in the current directory and then the system directories for the file
 - angle brackets (<>), it looks only in the system directories (e.g. “/usr/include”)
- The lines in an #included file are literally copied in place of the #include

#include Example

```
#include <stdio.h>  
  
#include <stdlib.h>  
  
#include <string.h>  
  
#include "mylibrary.h"
```

NOTE:

*These lines do NOT end in semicolon ;
They are NOT part of the C language
Rather, they are pre-processor directives*

*Header files typically contain
Macro definitions
Type Declarations
Function Prototypes
NOT Code!!!*

Macro Processing

- Macro processing is just text substitution using some very specific rules
- We're going to need to want to use symbolic names for constants in several places

- For example

```
#define NUL_CHAR '\0'  
#define MAXWORDLEN 256
```

*This is how we typically do constants in C.
Use a #define MACRO*

*(Macros are often ALL_CAPS by convention
But that is not required)*

- These symbolic names are **textually** replaced in the source code before it is compiled!
- If you invoke the compiler with **gcc -E**, it will show you the pre-processed input file with all the includes and macros expanded

Macro Example

- Continuing with our last example, we can write

```
char buf[MAXWORDLEN];  
...  
while (buf[i] != NUL_CHAR && i < MAXWORDLEN)  
    i++;
```

- After the C Preprocessor, the code literally becomes

```
char buf[256];  
...  
while (buf[i] != '\0' && i < 256)  
    i++;
```

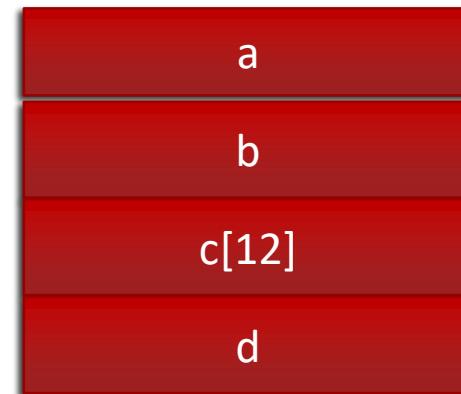
Macros with Arguments

- ↗ You can also use macros like functions (sort of).
 - ↗ `#define PRODUCT(a,b) a*b`
 - ↗ `PRODUCT(4,5)` would expand to `4*5`
- ↗ But what about `PRODUCT(x+3,y+4)` ?
 - ↗ Would expand to: `x+3*y+4`
 - ↗ Does that evaluate to what you intended?
- ↗ Solution: fully parenthesize (order of operation):
 - ↗ `#define PRODUCT(a,b) ((a)*(b))`
 - ↗ `PRODUCT(x+3,y+4)` now expands to `((x+3)*(y+4))`
- ↗ REMEMBER: `#define` is a basically a text search-and-replace operation on your source code file (i.e., it's not that smart).

Tip: There can't be a space between the macro name and the left paren!

- Structs look a lot like classes in Java – they are aggregate data types
- They contain no methods
- All data members are publicly visible

```
struct a {  
    int a, b;  
    char c[12];  
    int d;  
};
```



24 bytes – assuming `sizeof(int)` is 4 bytes

- The struct tag *declares* the type

```
struct car {  
    char mfg[30];  
    char model[30];  
    int year;  
};
```

NOTE:

*The type is 'struct car'
Not just 'car'*

- Names following the struct tag *defines* instances of the struct (i.e. allocate storage)

```
struct car mikes_car, joes_car;
```

- Identifier/function names and struct tags occupy separate name spaces. Struct members occupy a name space unique to the containing struct. That means that “struct b { int b} b;” is legal and not ambiguous.

- You can *declare* a struct type, and *define* an instance (or multiple instances) of the variable at the same time.

```
struct car {  
    char mfg[30];  
    char model[30];  
    int year;  
} mikes_car;
```

- struct car is a data type
- mikes_car is a variable of type struct car

Referencing Structure Members

- The . operator is used just as in Java to reference members of a structure

```
printf("%s\n", mikes_car.model);  
johns_car.mfg = "Chevrolet";
```



NOTE:

*You can't assign a string literal to an array like this in C.
You can use strcpy() - a library function - and you must
make sure your destination has room to hold the source string.*

Don't forget to #include <string.h>

Referencing Structure Members

- The . operator is used just as in Java to reference members of a structure

```
printf("%s\n", mikes_car.model);  
strcpy(johns_car.mfg, "Chevrolet");
```

- Can do the same things you can do with addresses in an assembly language
- Powerful and potentially dangerous of course
- No runtime checking (for efficiency)
- Java avoids many of the features of pointers that cause problems, hence the decision to call them references. In Java:
 - No “address of” or “dereference” operator
 - No pointer arithmetic
 - No casting of integer types to pointer types

- Pointers contain memory addresses (or NULL)
- Remember this from LC-3 assembly?

B	.fill	29
BADDR	.fill	B

- In C, we can write it

```
int b = 29;  
int *baddr = &b;
```

- & is the “address of” operator; you use it to obtain a pointer to an existing data item

Using Pointers

- To reference what a pointer points to, we use the dereference operator *
- It acts like “indirect” in LC-3 loads and stores

- In LC-3 assembly, if we wanted to use BADDR to access the value in B, we used

```
LDI R1,BADDR ; get B  
ADD R1,R1,#2  
STI R1,BADDR ; set B=R1
```

- In C, we can write this as
`*baddr = *baddr + 2;`
- It literally does the same thing!

Another Example

- Continuing our example with **b** and **baddr**,

```
printf("%d\n", b);  
*baddr = 18;  
printf("%d\n", b);  
printf("%d\n", *baddr);
```

prints

```
29  
18  
18
```

- The value stored at **b** has been changed to 18; **baddr** still contains the address of **b**

* Has Two Meanings!

- Don't let the * in declarations confuse you about executable statements.
- In a type declaration, * means "pointer to" and is part of the type
- In an expression, * is the "dereference" operator – it acts like the "indirect" in the LC-3 loads and stores
- When applied to a pointer, * makes the expression mean "What this pointer points to"

Declaration

```
int i, x;
```

```
int *px = &x;
```

Expression

```
i = *px;
```

Question

Given the definition and initializer

`int *ptrx = *y;`

what must the type of `y` be?



- A. pointer to a pointer to an int
- B. pointer to an int
- C. int
- D. None of the above

Today's number is 31,821

(declaration) = (expression)
`int *ptrx` = `*y;`

`ptrx` is a "pointer to integer" so
the expression `*y` must yield a
type of "pointer to integer"

If we guess `y` is "pointer to
integer", then dereferencing with
`*y` would yield type "integer". No
good.

If we guess `y` is a "pointer to
pointer to integer", then `*y` would
yield type "pointer to integer".
That's what we need, so earlier
in the code, we should see

`int **y;`

Pointers and Arrays

```
int a[10];
```

```
int *p;
```

- a is an array, size 10, of int
- p is a pointer to int (memory address)

- In an expression, an array name becomes a *constant* pointer to the first element
 - e.g., a without [brackets]
 - a is the memory address where the array starts
 - a can't be changed (but its contents can)
 - In other words, it's the address of a[0] (or specifically &a[0])

Pointers and Arrays

```
int a[10];
int *p;

// legal
p = a;          // p is the memory address of
                // the start of the array

// illegal
a = p;          // a is a CONSTANT pointer
                // you are not allowed to change its value
```

Strings and pointers

```
char str[6] = "Hello";
```

```
char *s;
```

- ↗ Recall: a C string is really an array of char
- ↗ When you see “char *”, think string (*for now*)

Pointer Arithmetic

- ↗ You can add or subtract pointers and integers
 - ↗ a pointer plus or minus an integer yields a pointer
- ↗ Just like adding to addresses on LC-3
 - ↗ with one additional semantic
- ↗

```
int *p = &i;  
p = p + 1;  
is interpreted as  
p = p + 1 * sizeof(*p);
```
- ↗ In other words, if p is an *int* pointer, $p + 1$ is the address of the **next int** – the address in p is incremented by the size of an *int*

↗ For example:

- ↗

```
int b[3] = { 9, 12, 13 };  
int *p = &b[0];
```
- ↗ The value of $*p$ is 9, $*(p + 1)$ is 12, and $*(p + 2)$ is 13
- ↗ **In an expression, $a[i]$ means exactly the same as $*(a + i)$**
- ↗ This is why the value of $p[0]$ is 9, $p[1]$ is 12, and $p[2]$ is 13, just as above!

Pointer arithmetic and Arrays

```
int a[10];  
int *p;  
p = a;
```

These expressions are equivalent.

(an int)

a [5] \longleftrightarrow * (a+5)

p [5] \longleftrightarrow * (p+5)

These expressions are equivalent.

(pointer to an int)

&a [5] \longleftrightarrow a+5

&p [5] \longleftrightarrow p+5

Question?

```
int x = 3;  
  
int y = 72;  
  
int *px = &x;  
  
int *py = &y;  
  
*px = 7;  
  
py = px;  
  
x = 12;  
  
printf("%d %d\n", *px, *py);
```

a) 3 72

b) 72 3

c) 7 12

d) 12 7

e) 3 3

f) 72 72

g) 12 12 ←

h) 12 72

i) 72 12

x	y	px	py
3			
	72		
		&x	
			&y
7			
			&x
12			

*px and *py both evaluate to 12
since they now both point to x!

What is the output?

Pointers and Structs

- ☞ C supports the operator `->`
- ☞ The left operand must be a pointer to struct and the right operand a **struct** member.
- ☞ It is literally a shorthand for `*` and `.`

```
struct myStruct {  
    int a, b;  
} *p; // p is a pointer to struct myStruct
```

```
(*p).a = (*p).b;  
p->a = p->b;
```

Same
meaning

```
int mult(int a, int b) {  
    return a*b;  
}
```

- Two arguments (int), pushed on the stack
- Returns an int

- Pass by value always (not by reference)
- Copies of arguments are pushed on the stack

Functions: main()

```
int main(int argc, char *argv[]) {  
    return 0;  
}
```

- main() is just another function
- It is the first function invoked when you run your program from the command line
- Convention: return 0 on success, or non-0 on error
- Hint: type echo \$? in the Linux shell to see the return value from main()

Functions: void

```
void func(void) {  
    printf("Hello\n");  
}
```

- void is a special keyword
- Return type void:
 - *function does not return a value*
- Arguments of void:
 - *function takes no arguments*

Functions: arrays as arguments

```
char s[10] = "Hello";  
  
void test(char *s) {  
    printf("%s\n", s);  
}  
  
int main(int argc, char *argv[]) {  
    test(s);  
    test(&s[0]);  
}
```

Same
meaning

- ↗ To pass an array into a function, we pass a pointer to the first element
- ↗ Note that an array name in an expression (function call) is automatically promoted to a pointer to its first element
- ↗ This includes strings (arrays of char)
- ↗ We did the same thing in assembly

Declare functions before you call them

```
int foo(int);  
int bar(char *);
```

Function
Prototypes

```
int main() {  
    foo(1);  
    bar("Bye");  
}
```

```
int foo(int i) {  
    // Do stuff  
}
```

Function
Definitions

```
int bar(char *s) {  
    // Do stuff  
}
```

We often put the main() function first

- But main() needs to know about the functions it will call
- And they are defined below

So we add function declarations above main()

- They just have the name, ret type, and args
- And end in a semicolon
- These are "function prototypes"

Command Line Arguments

```
int main(int argc, char *argv[]) {  
    return 0;  
}
```

- ↗ argc is the number of arguments
 - ↗ Including the program name itself
 - ↗ *Think “argument count”*
- ↗ argv is an array of strings
 - ↗ More properly, an array of pointer to char
 - ↗ *Think “argument values”*

Command Line Arguments

```
int main(int argc, char *argv[]) {  
    return 0;  
}
```

If we call our program this way on the command line

\$./myprogram cs2110 rocks

then

argc == 3

argv[0] is “./myprogram”

argv[1] is “cs2110”

argv[2] is “rocks”

A Quick Example: args.c

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    for (int i=0; i<argc; i++) {
        printf("arg #%-d: '%s' \n", i, argv[i]);
    }
}
```

Running Our Example

```
$ gcc args.c  
$ ls  
args.c          a.out  
$ ./a.out hello there  
arg #0: './a.out'  
arg #1: 'hello'  
arg #2: 'there'  
$
```

*The default name of an executable program from gcc is:
a.out*

You can rename the executable with the -o flag
`$ gcc -o args args.c`
// the executable will now be named args
Or you can use mv to change its name:
`$ mv a.out args`

Examples: IntroToC

- ☞ Download the IntroToC folder from Canvas>Files
 - ☞ args.c
 - ☞ strings.c
 - ☞ structs.c
 - ☞ segfault.c
- ☞ Read the comments
- ☞ Compile and run the code
- ☞ Play with the code.
 - ☞ Try to do other similar things
 - ☞ Try to break it
- ☞ Look at the next slide deck and try out **gdb** to trace through the code

Continuing with C

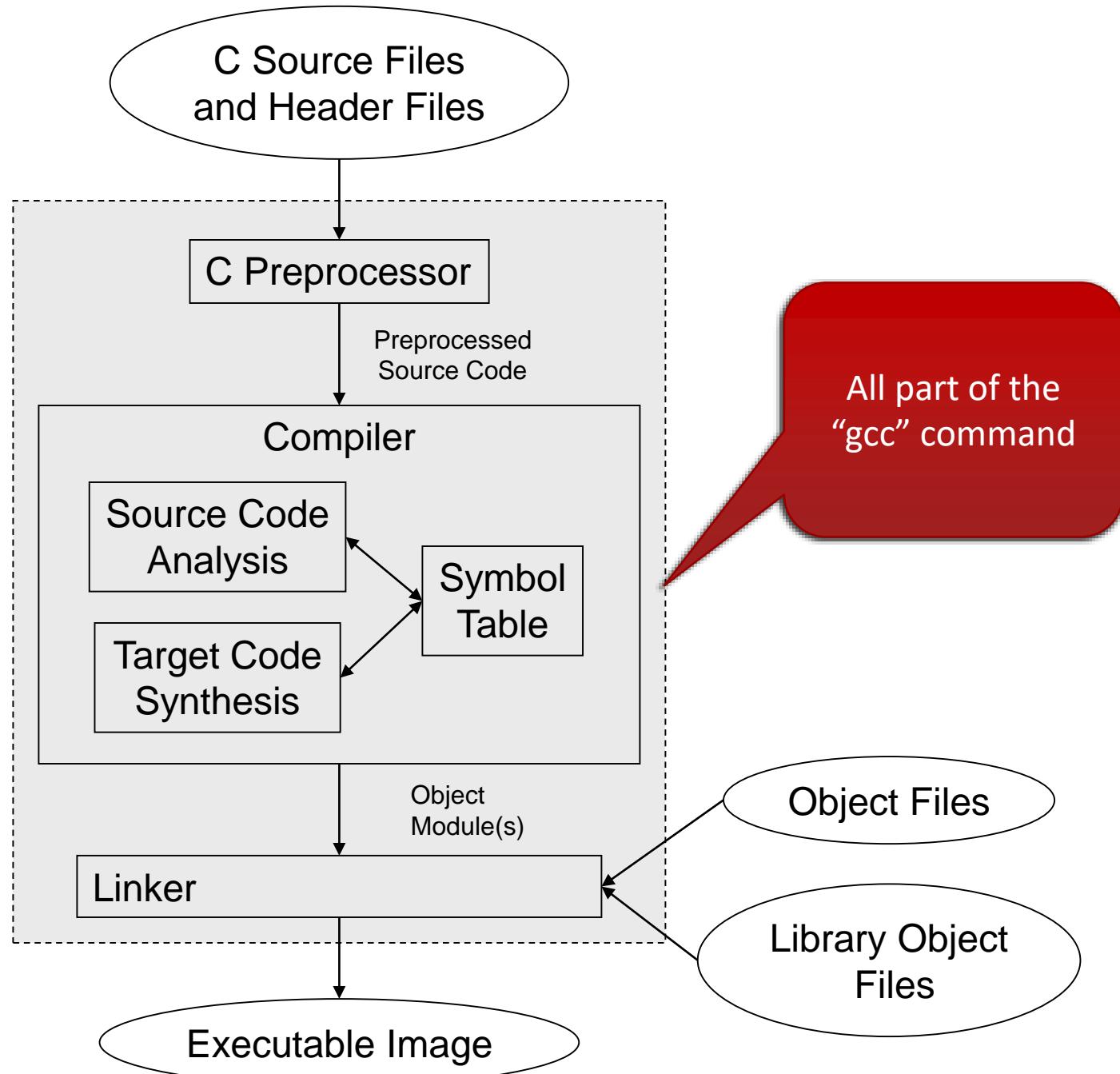


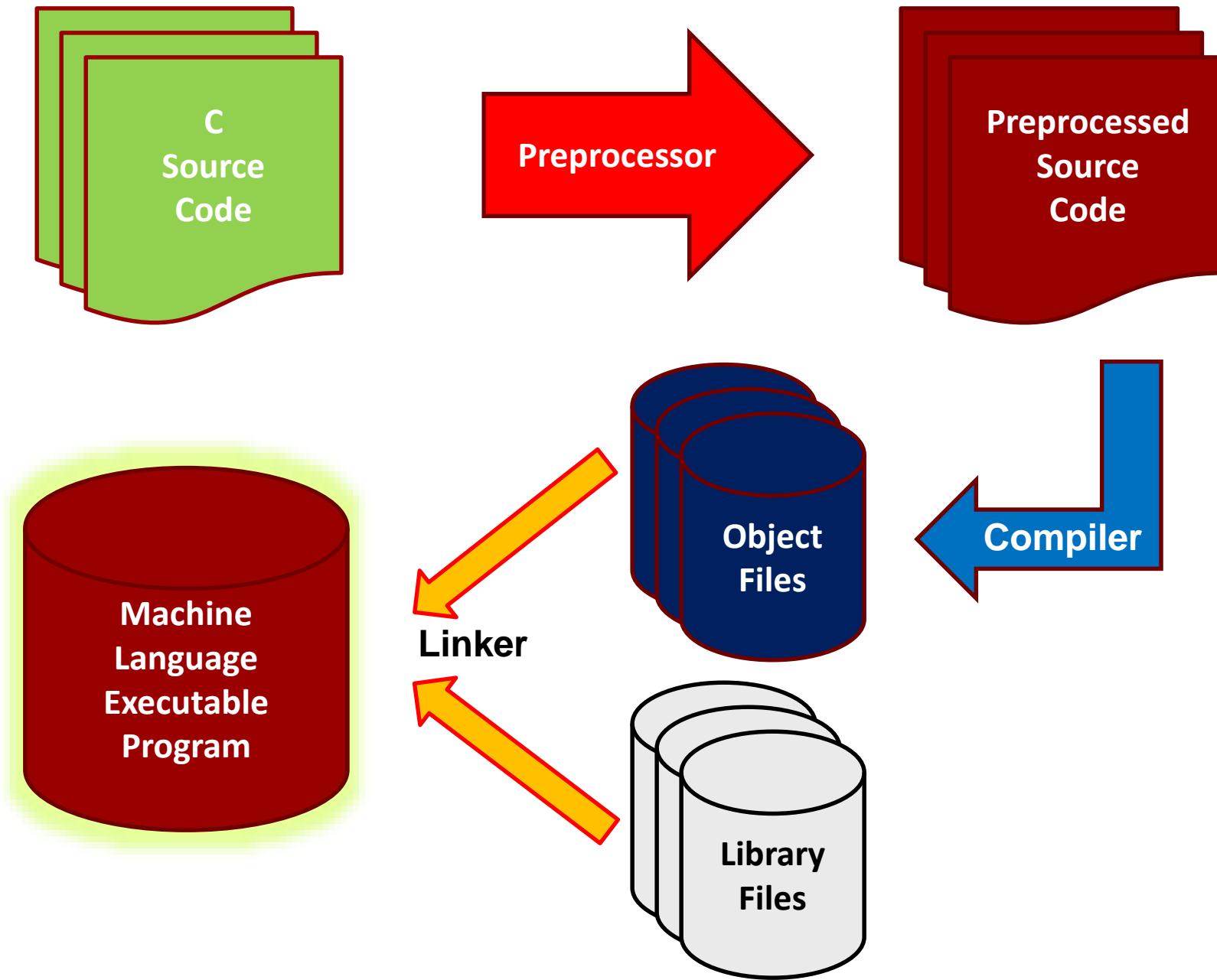
- C Preprocessor: Compiling, Linking, Execution
- Storage location
 - Memory Layout
 - Storage Classes and Type Qualifiers
 - Scopes
- C Type Declarations
- C Functions
 - Pass by value, implementing pass by reference

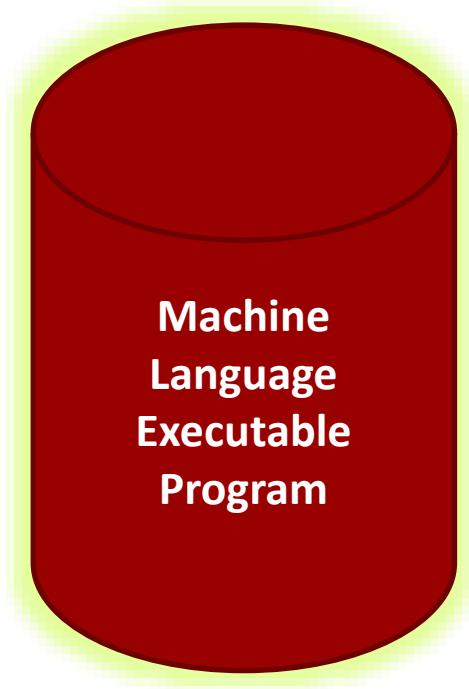
C Preprocessor: Compiling, Linking, Execution



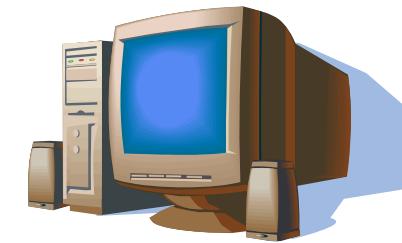
The C Compiler







Loader (Linux)

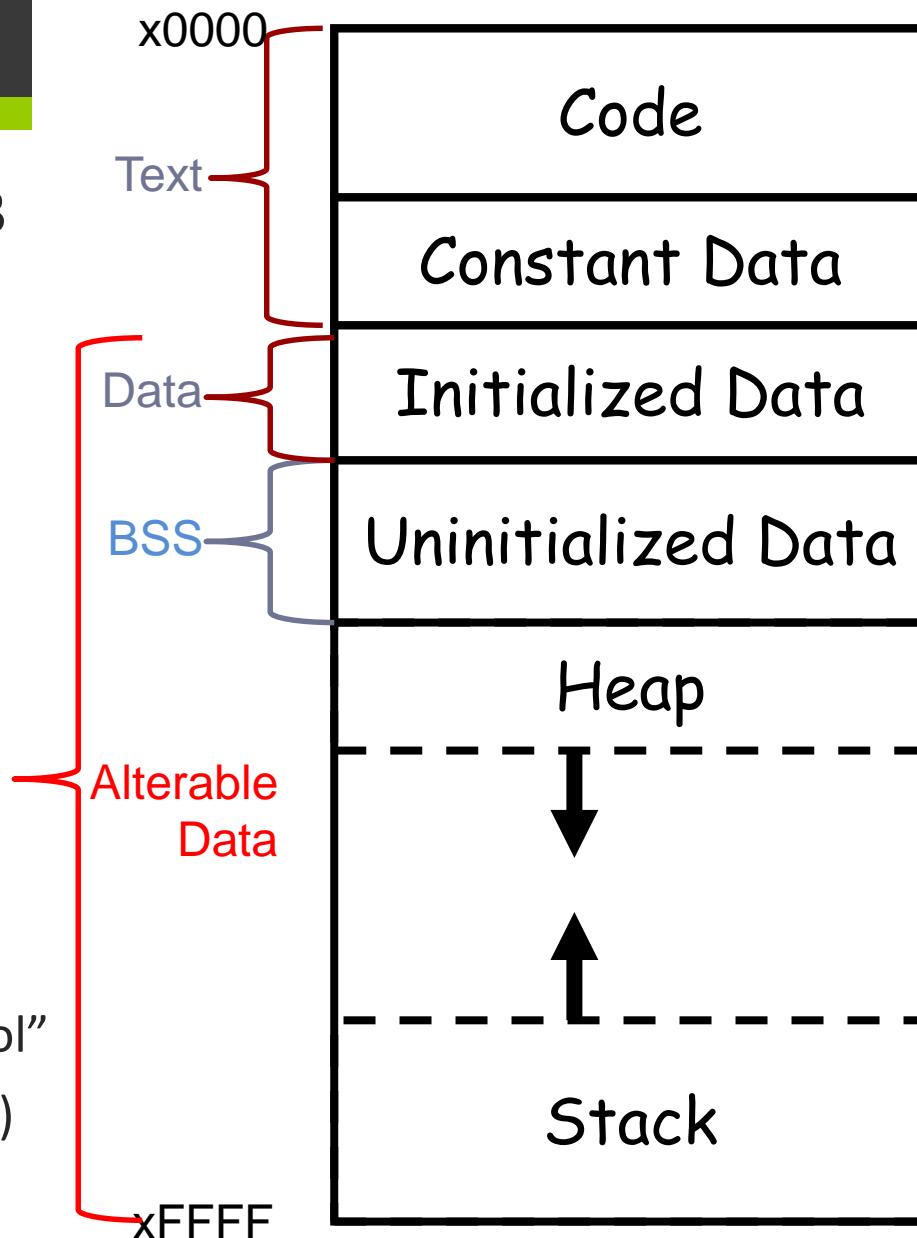


Data Storage



Memory Layout

- The traditional C executable has 3 segments:
 - Text
 - Program code (unmodifiable)
 - Constant data (maybe)
 - Data
 - Constant data (maybe)
 - Initialized variables
 - BSS
 - Uninitialized static data
 - From “Block Starting with Symbol”
 - (in other words, .blkw from LC-3)



- ↗ File – seen by the entire file, after the definition
 - ↗ Outside a function definition
 - ↗ Global - The keywords **extern** and **static** control whether the name is seen in *other* files

<no keyword>	External definition – visible to other files – defines storage
static	Visible to no other files
extern	External reference – visible to other files – declares reference to an external definition elsewhere

- ↗ Block – seen within the block, after the definition

Global Scope Example

file1.c:

```
int a[10];          // external definition
static struct r *p; // this file only
extern float c[100]; // ref to c in file2.c
...

```

file2.c:

```
extern int a[10];    // ref to arr in file1.c
static struct r *p; // NOT the p above
float c[100];       // definition
...

```

Block Scope Example

```
#include <stdio.h>
int i = 0;
void f(int);

int main(int argc, char *argv[] ) {
    i = 1;
    int i = 0;
    i = 2;
    for (int i= 0; i < 10; i++) {
        f(i);
        int i = 2;
        f(i);
    }
    f(i);
}
void f(int j) {
    printf("%d\n", j);
}
```

The diagram illustrates variable scope and shadowing in C. It shows the declaration of 'i' at the top level and its use in the assignment 'i = 1'. A red arrow points from this declaration to the assignment. Inside the main function, there is another declaration of 'i' with a value of 0. A red arrow points from this declaration to the assignment 'i = 2'. Within the for loop, there is a third declaration of 'i' with a value of 2. A red arrow points from this declaration to the two 'f(i)' calls. Finally, outside the for loop, there is a fourth declaration of 'i' with a value of 2. A red arrow points from this declaration to the final 'f(i)' call.

Storage Classes and Type Qualifiers

Storage Class Specifiers

register

auto

static

extern

const

volatile

restrict

Type Qualifiers

Storage Class

- ↗ Storage class tells us **where the data will be stored** and **who will be able to see it**
- ↗ It is NOT part of the type; it affects only the variables defined or declared using it
- ↗ The rules for storage classes are NOT regular; you will need to memorize them or have a ready reference

Storage Class

	<i>Outside a function definition</i>	<i>Inside a function definition</i>
<none>	scope: external definition storage: static address	scope: within the function storage: on the stack

Storage Class

	<i>Outside a function definition</i>	<i>Inside a function definition</i>
<none>	scope: external definition storage: static address	scope: within the function storage: on the stack
auto	N/A	scope: within the function storage: on the stack

same

Storage Class

	<i>Outside a function definition</i>	<i>Inside a function definition</i>
<none>	scope: external definition storage: static address	scope: within the function storage: on the stack
auto	N/A	scope: within the function storage: on the stack
static	scope: within the file only storage: static address	scope: within the function storage: static address

same

Storage Class

	<i>Outside a function definition</i>	<i>Inside a function definition</i>
<none>	scope: external definition storage: static address	scope: within the function storage: on the stack
auto	N/A	scope: within the function storage: on the stack
static	scope: within the file only storage: static address	scope: within the function storage: static address
extern	scope: external reference storage: static address; location determined by file containing the external definition (can't have initializer either)	scope: external reference storage: static address; location determined by file containing the external definition (can't have initializer either)

same

Storage Class

	<i>Outside a function definition</i>	<i>Inside a function definition</i>
<none>	scope: external definition storage: static address	scope: within the function storage: on the stack
auto	N/A	scope: within the function storage: on the stack
static	scope: within the file only storage: static address	scope: within the function storage: static address
extern	scope: external reference storage: static address; location determined by file containing the external definition (can't have initializer either)	scope: external reference storage: static address; location determined by file containing the external definition (can't have initializer either)
register	N/A	scope: within the function storage: register or stack (hint to compiler; use of & not allowed; seldom used)

same

static inside a function

```
void funcCounter(void) {  
  
    static int counter = 0;           // one instance, static memory  
  
    counter++;                     // visible only in function  
    printf("This function was called %d times.\n", counter);  
  
}
```

- ↗ counter is allocated once in the Initialized Data static area of memory
- ↗ counter is NOT located on the stack
- ↗ Every call to funcCounter() sees the same shared instance of counter.
- ↗ It is only initialized to 0 once at program load (not every time function is called).
- ↗ It's the analog of .fill from LC-3

- ↗ What would happen if we did NOT use static?

Type Qualifier

- ↗ Part of the type (unlike Storage Class)
- ↗ Not mutually exclusive with a Storage Class
- ↗ **const** – the value of this variable may not be changed after initialization
- ↗ **volatile** – the compiler may not optimize references to this variable (e.g. it's a device register that may change value asynchronously)
- ↗ **restrict** - For the lifetime of a pointer, only the pointer itself or a value directly derived from it may be used to access the object to which it points. This allows better optimization.

Most important to remember

- ↗ static: has two meanings
 1. *Inside a function*: static changes the storage location to static memory, either data or BSS segment (the scope stays local)
 2. *Outside a function*: static changes the scope (visibility) to be only visible within the file (the storage location stays in static memory)
- ↗ extern
 - ↗ Compiler does NOT allocate storage
 - ↗ For type checking of the identifier name only
 - ↗ Another C file must allocate storage by defining that var name (or function)
- ↗ volatile
 - ↗ Tells the compiler not to optimize away the variable
 - ↗ Use this for device registers

Question

You encounter the following definition **before the first function definition** in a file. What is its scope and where is F stored?

```
double F;
```

- A. Scope is all functions in the file; stored on the stack
- B. Scope is all functions in all files; stored in static memory
- C. Scope is the first function in the file; stored in static memory
- D. Scope is all functions in all files that include a corresponding **extern** declaration; stored on the stack
- E. Scope is all functions in all files that include a corresponding **extern** declaration; stored in static memory



Question

What is the type of p:

```
static const int *p;
```

- A. pointer to int
- B. static pointer to const int
- C. static pointer to int
- D. pointer to const int

Remember:

- Storage classes are not part of the type
- Type qualifiers ARE part of the type



Understanding C type declarations

- A **declaration** in C introduces an identifier and describes its type, be it a scalar, array, struct, or function. A declaration is what the compiler needs to accept references to that identifier (for type checking).

You may have as many declarations of an identifier as you want within a scope as long as they are consistent.

- A **definition** in C actually instantiates/implements this identifier. For instance, a definition allocates storage for variables or defines the body of a function.

You may only have one definition of an identifier within a scope.

C Type Declarations Have Two Parts!

- First comes the **Base Type**
 - This is the type (or struct s or a typedef), and optionally a storage class and/or a type qualifier
 - The Base Type applies to all names up until the semicolon
- Then comes the list of **Declarators**, separated by commas
 - Each of these declares a type for each identifier (the variable name)
 - Each is based on the Base Type, but stands alone
 - Consists of sensible combinations of “pointer to”, “array of”, or “function returning”, ending with the Base Type

Declaration Example

```
static volatile long int i, *j, k[10];
```



Base type: volatile long int

(with type qual and storage class)

Declarator: no modifiers

i is type **volatile long int**

Declarator: "pointer to"

j is type **pointer to volatile long int**

Declarator: "array[10] of"

k is type **array[10] of volatile long int**

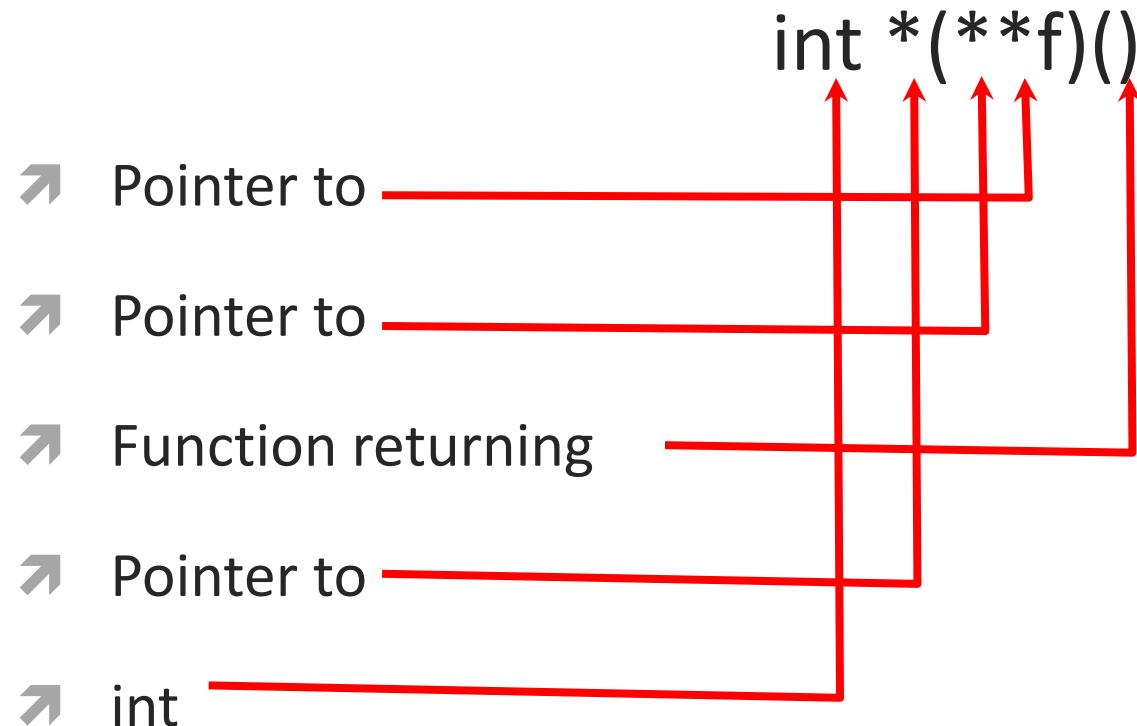
Each declarator is separate

- ↗ int *x, y, z;
- ↗ x is a pointer to int
- ↗ What is y?
 - ↗ y is NOT a pointer to int
 - ↗ y is just an int
 - ↗ The * goes with each identifier (not with int)
 - ↗ Each declarator is separate, and references the base type (int)
- ↗ If we want all of these to be type “pointer to int”, write:
- ↗ int *x, *y, *z;

Reading and Forming Declarators

- ↗ C declarations combining pointers, arrays, and functions aren't mysterious...
- ↗ **Rule 1:** Remember the precedence of the declarators
 - ↗ () and [] declarators get processed first
 - ↗ * gets processed last
 - ↗ Parentheses change the precedence order (just as in expressions)
- ↗ **Rule 2:** Read or form the declarations from the **inside out**
- ↗ Example: int *(*f)()
 - ↗ f is a pointer to a pointer to a function returning a pointer to int.
- ↗ So how do we come to that answer?

Using Our Example



How Do We Do It Automatically?

- ↗ **parsing**: determining which grammar productions (rules) were used to generate a sentence

Look at the grammar in K&R appendix A, p.235, for the ***declarator*** and ***direct-declarator*** productions

- ↗ Here's simple demo that's hard-coded
 - ↗ See K&R 5.12
 - ↗ Grab dcl.c from Canvas>Files>Source Code
 - ↗ Then try it...

Output from dcl.c

```
$ gcc dcl.c
$ ./a.out
int b()
b: function returning int
int c[1]
c: array[1] of int
int *c[1]
c: array[1] of pointer to int
int (*c)[1]
c: pointer to array[1] of int
int *silly()
silly: function returning pointer to int
int **silly()
silly: function returning pointer to pointer to int
int *(*silly)()
silly: pointer to function returning pointer to int
int *(**f)()
f: pointer to pointer to function returning pointer to int
```

Declaring vs. Using Pointers and Arrays

- Recall that *, (), and [] have different meanings in declarations and executable statements
- Conveniently, their behavior allows you to “unwind” a type just as the declaration would wind it
- Take our previous example, `int *(**f)()`
 - We called it Pointer to Pointer to Function returning Pointer to int
 - If want to “unwind” it, just apply the operators in the order they are named
- To remove the first “Pointer to”, use the * (dereference) operator
 - `*f` is a Pointer to Function returning Pointer to int
- Now remove the second “Pointer to” with another *
 - `**f` is a Function returning Pointer to int
- Now to remove the “Function returning”, we need to call the function
 - `(**f)()` calls the function and returns type “Pointer to int”
- Now, since we have a pointer, if we want to extract the int value, use * again!
 - `*(**f)()` is indeed of type int
- Note the similarity between the declaration and the expression accessing the content! This is not an accident!

Declaring vs. Using Pointers and Arrays

- ↗ char *b(); // function returning pointer to char
 - ↗ char *bp = b(); // pointer to char
 - ↗ char bc = *b(); // char
- ↗ double **d; // pointer to pointer to double
 - ↗ double *pd = *d; // pointer to double
 - ↗ double dv = **d; // double
- ↗ int *(*c)[]; // ptr to array of pointers to int
 - ↗ int *apc[] = *c; // array of ptrs to int
 - ↗ int *pc = (*c)[1]; // pointer to int
 - ↗ int cv = *(*c)[1]; // int

Question

You are given the C definition below. What is its type?

```
double **m[][];
```

- A. Pointer to pointer to array of array of doubles
- B. Pointer to array of pointers to array of doubles
- C. Array of array of pointers to pointers to doubles
- D. Array of pointers to array of pointers to doubles



Typedef

- Typedef is a shortcut that allows you to create a new alias for a type
- It does **not** create a new type
- Start the declaration with “typedef” and put the alias name where you would put the variable name
- Don’t let the syntax throw you: It just allows you to create another name for an existing type

Typedef Examples

- ↗ Define an array of 5 struct a named b:
`struct a b[5];`
- ↗ Create a type alias for an array of 5 struct a
`typedef struct a sa5[5];`
- ↗ Now we can use sa5 as a type name:
`sa5 c;`
- ↗ Note that *b* and *c* are the **same type!**
- ↗ Another example
`typedef unsigned long size_t;
size_t position;`
- ↗ What type is *position*?
`unsigned long`

Function Calls



- ↗ Functions must be declared before they are used
- ↗ Hence the need for function prototypes, which are basically a function header without a body, e.g.

```
int nfact(int);
```
- ↗ Recall that a function prototype is a declaration; with a body it's a definition
- ↗ A function prototype is automatically given the **extern** storage class
- ↗ Recall you may have any number of declarations as long as they're identical; you may only have ONE function definition
- ↗ Parameters are **always** passed as **call-by-value**
 - ↗ Copies of the parameters (even structs) are pushed on the stack
 - ↗ Arrays turn into pointers to their first element, a copy of which is passed
- ↗ You can implement **call-by-reference** parameters easily with pointers

- Assignment: Write a function that will swap two integers

- First try:

```
void swap(int a, int b)
{
    int t;
    t = a;
    a = b;
    b = t;
}
```

Anything wrong?

- ↗ We call it like this

```
int x = 42;
```

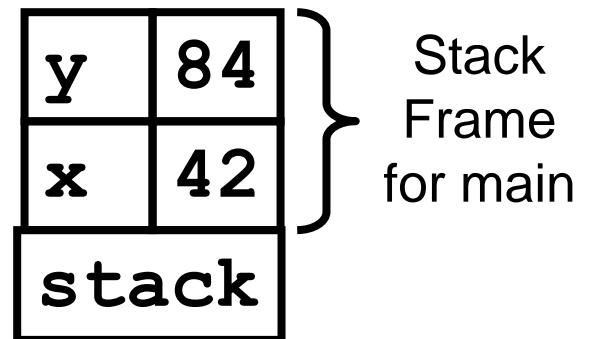
```
int y = 84;
```

```
swap(x, y);
```

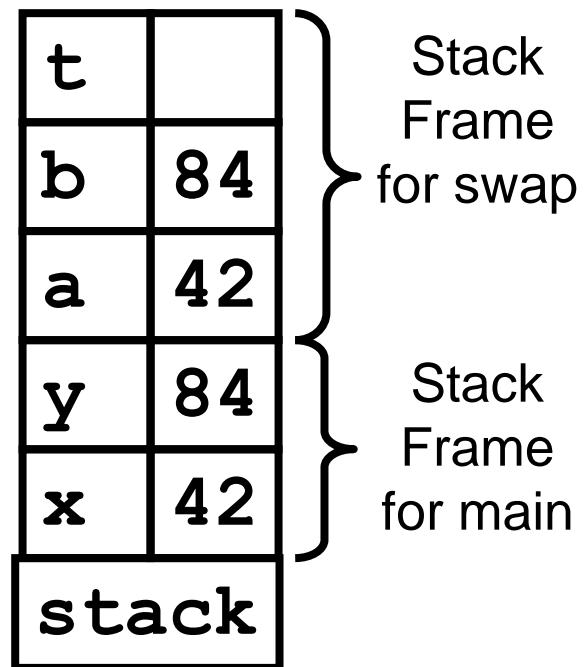
```
int x = 42;  
int y = 84;  
swap(x, y);  
  
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```

Note that we're NOT including the **register save** areas and other machine-dependent details in these stack frame visualizations; we show only the arguments and local variables.

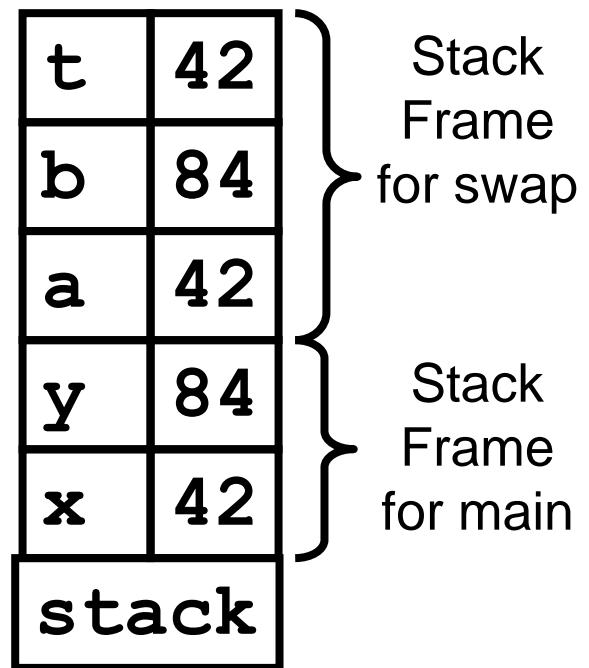
C handles the details of the stack from out of your sight.



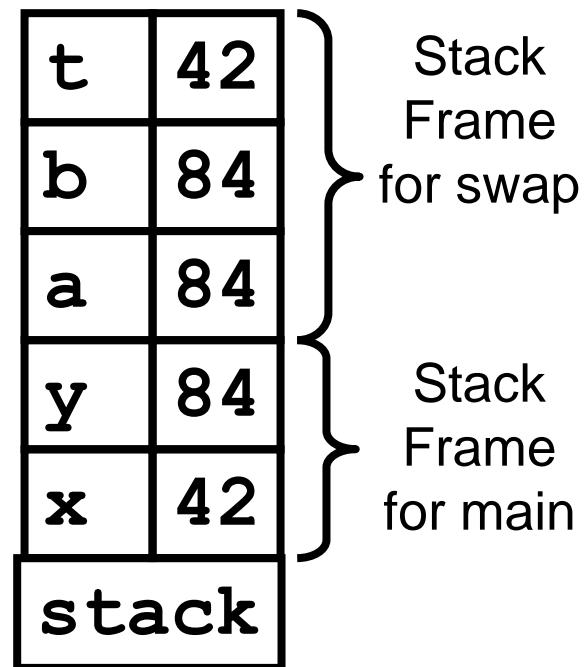
```
int x = 42;  
int y = 84;  
swap(x, y);  
  
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```



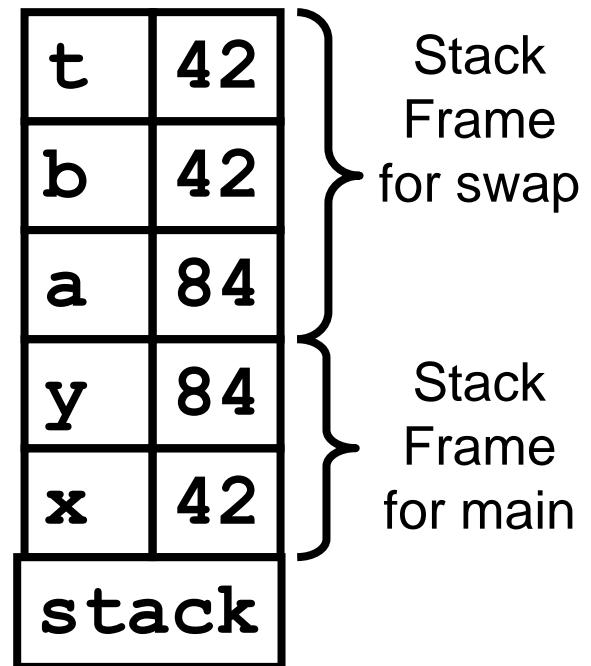
```
int x = 42;  
int y = 84;  
swap(x, y);  
  
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```



```
int x = 42;  
int y = 84;  
swap(x, y);  
  
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```



```
int x = 42;  
int y = 84;  
swap(x, y);  
  
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```



```
int x = 42;  
int y = 84;  
swap(x, y);
```



What are x and y?

```
void swap(int a, int b)  
{  
    int t;  
    t = a;  
    a = b;  
    b = t;  
}
```

pop

y	84
x	42
stack	

Stack
Frame
for main

The True Way

- ↗ To swapping happiness

Swap for Real This Time

- ↗ Assignment: Write a function that will swap two integers
 - ↗ This is how we do pass-by-reference in C.
- ↗ Correct implementation:

```
void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Now it works...

↗ We call it like this

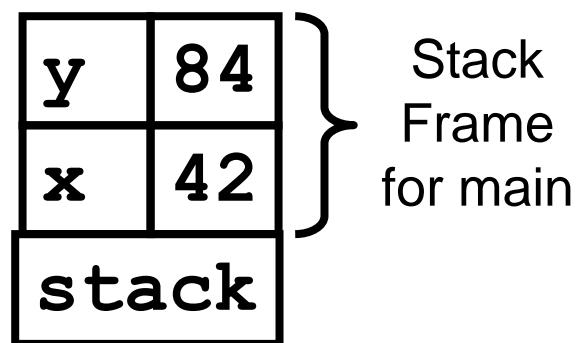
```
int x = 42;
```

```
int y = 84;
```

```
swap(&x, &y);
```

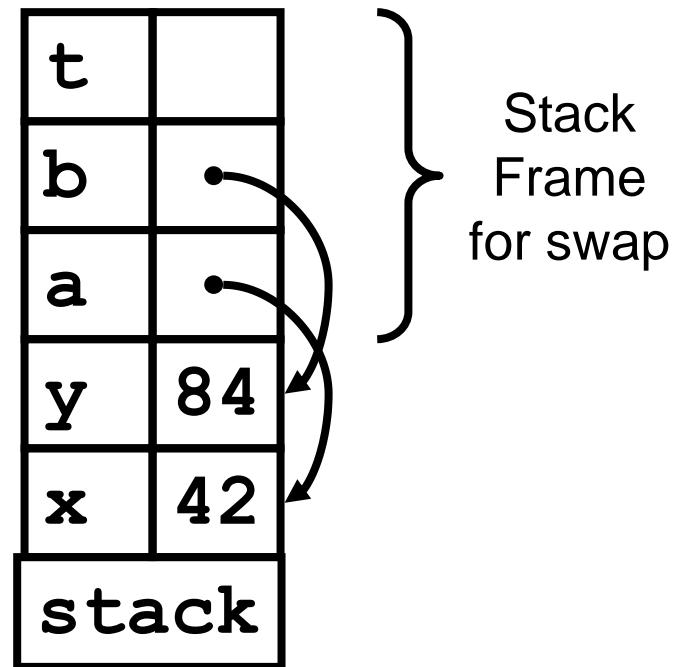
```
int x = 42;  
int y = 84;  
swap(&x, &y);
```

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```



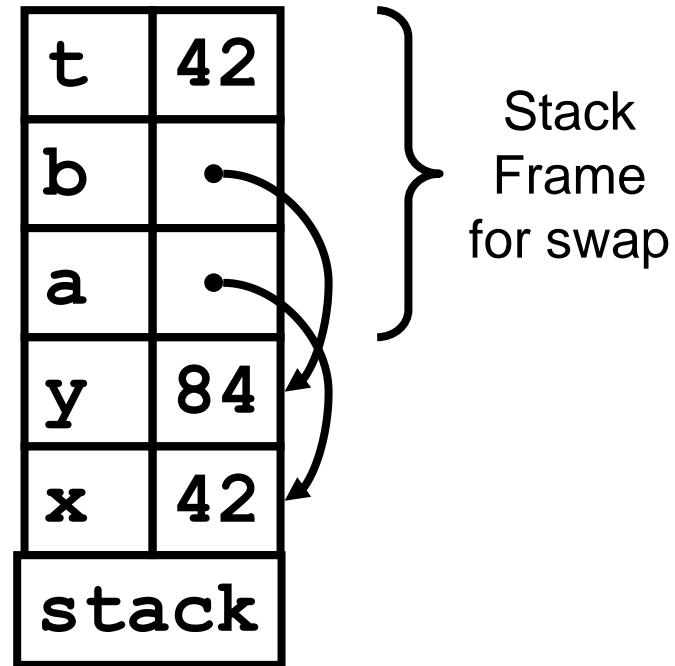
```
int x = 42;  
int y = 84;  
swap(&x, &y);
```

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```



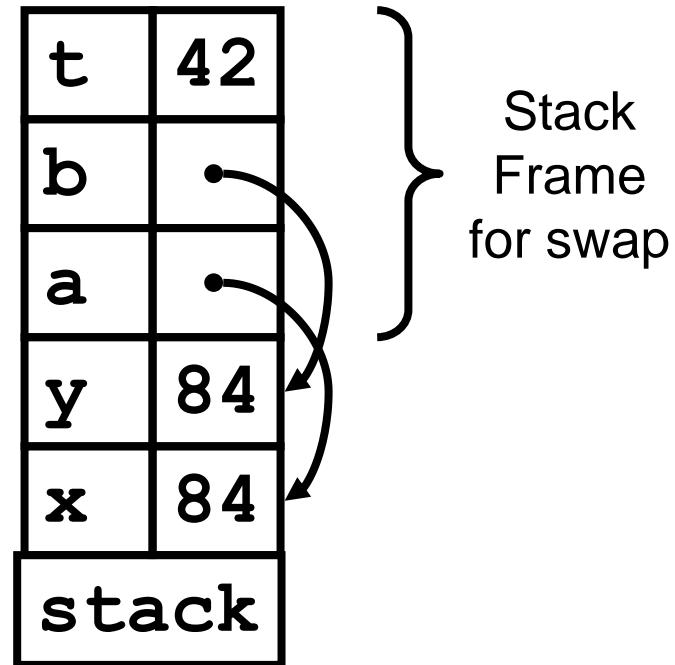
```
int x = 42;  
int y = 84;  
swap(&x, &y);
```

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```



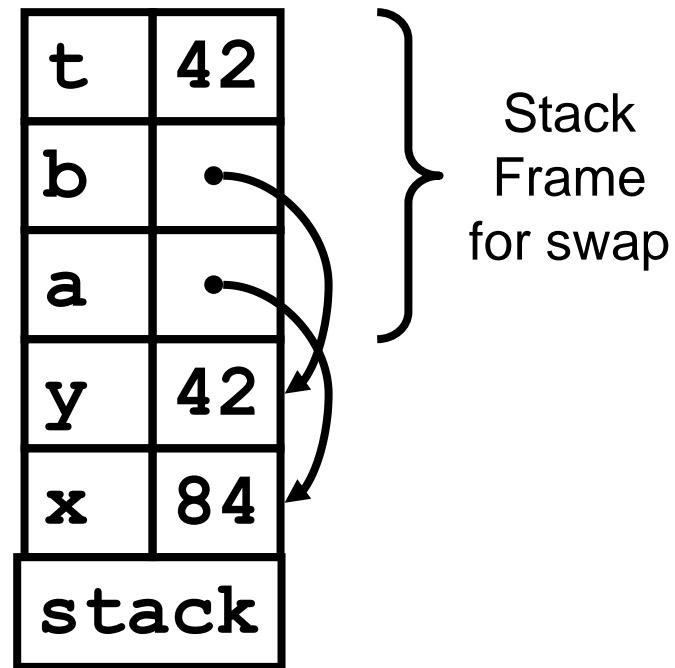
```
int x = 42;  
int y = 84;  
swap(&x, &y);
```

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```



```
int x = 42;  
int y = 84;  
swap(&x, &y);
```

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```



```
int x = 42;  
int y = 84;  
swap(&x, &y);
```



What are x and y?

```
void swap(int *a, int *b)  
{  
    int t;  
    t = *a;  
    *a = *b;  
    *b = t;  
}
```

pop

y	42
x	84
stack	

Stack
Frame
for main

Another Way of Calling It

```
int x = 42;
int y = 84;
int *px = &x;
int *py = &y
swap(px, py);

void swap(int *a, int *b)
{
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Question

Today's number
is 39,390

Parameters in C function calls are always passed by value

- A. except for structs which are passed by reference
- B. except for arrays which are passed by reference
- C. because array and struct arguments are copied into the stack
- D. recognizing that array arguments become pointers which are passed by value
- E. but in some sense, D and B are both true

```
int f(int p[]) {  
    p[1] = 12;  
}  
int g(int *p) {  
    p[1] = 12;  
}
```

Same meaning

```
int x[10];  
int *q = x;  
f(x);  
f(q);  
g(x);  
g(q);
```

Same results

Arrays



- ↗ Manual bounds checking
- ↗ How do we tell the length of an array?
- ↗ If it's defined in scope, use the idiom
`sizeof(ary) / sizeof(ary[0])`
- ↗ If it's passed as a parameter or in the heap
DIY in C: You must pass the length with the array
- ↗ For an array in the heap, you may need to keep both the allocated size of the array **and** the number of elements currently stored in the array
- ↗ C arrays have but a single dimension; however, C is happy to support arrays of arrays, etc.

Arrays (Like Pointers) Have Two Meanings

- ↗ Used in a **declaration or type**, an array in C describes a block of storage! The definition

```
short array[1026];
```

would assemble as

```
array .blkw 1026
```

- ↗ Arrays (and all C variables) are always allocated in a single contiguous memory block. So are arrays of arrays, etc.

```
short matrix[10][10][10];
```

would assemble as

```
matrix .blkw 1000
```

Arrays in Expressions

- ↗ Used in an expression, the array name acts as a ***constant pointer*** to the first element of the storage allocated for the array (as such we ***never, ever*** put & in front of an array name)
- ↗ Anything you can do with an array name, you can do with a pointer; you can even assign the name of an array of type-X to a pointer to type-X
- ↗ By the way, this notational convention prevents you from assigning one array to another in C! Why?
- ↗ ***Any array element selection you can do using subscripts can be done with pointers and pointer arithmetic*** because the subscript operator (square brackets) is literally a typographical shortcut for pointer arithmetic
 - ↗ *(p+n) is literally the same as p[n] (or more surprisingly n[p])

Recall

- Just use values in braces "{}"
`int ib[5] = { 5, 4, 3, 2, 1 };`
- The compiler will even count them for you
`int ib[] = { 5, 4, 3, 2, 1, 0 };`
- Characters initializers are similar
`char cb[] = { 'x', 'y', 'z' };`
- But you can arrange for special treatment
`char cb[] = "hello";`
- But note this is very different from
`char *cb = "hello";`

Notes about sizeof

- ```
char c1[] = "hello";
char *c2 = "hello";
```
- sizeof(c1) == ?
    - sizeof(c1) == 6 // array size, including the null terminator \0
  - sizeof(c2) == ?
    - sizeof(c2) == 8 // the size of a pointer on your system
    - Note: sizeof a pointer is implementation dependent
  - strlen(c1) == ?
    - strlen(c1) == 5 // same as strlen(c2)

# Recall

- The constant data area is where items such as the "Hello" in a statement such as

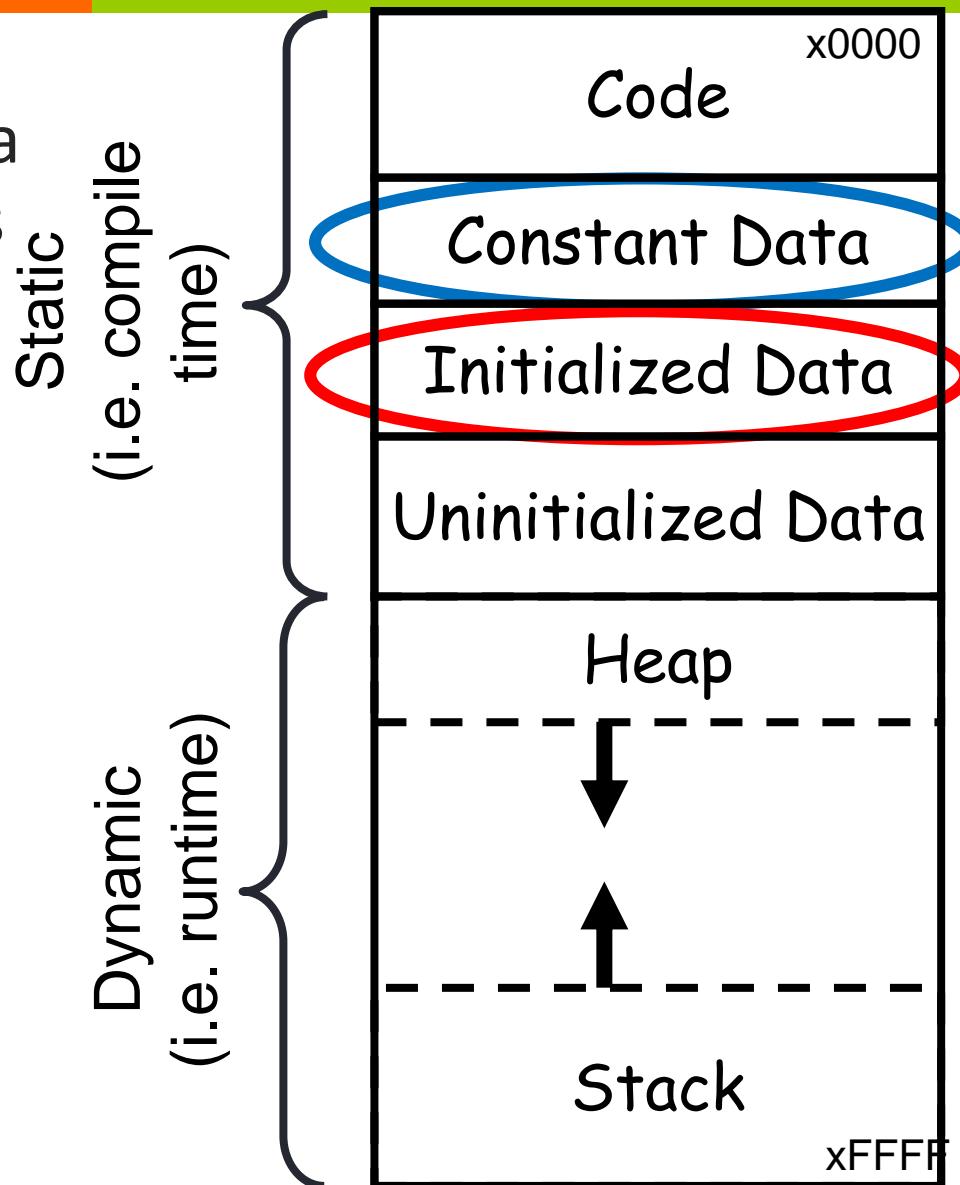
```
printf ("Hello");
```

would be stored.

```
char *cp = "Hello!";
```

```
char ca[] = "Hello!";
```

## Typical Arrangement



# Confused?

|                                       | Can pointer value be changed? | Can "Hello" be changed? |
|---------------------------------------|-------------------------------|-------------------------|
| <code>char *cp =<br/>"Hello";</code>  | Yes                           | No                      |
| <code>char ca[] =<br/>"Hello";</code> | No                            | Yes                     |

```
int ia[6];
```

- Allocates consecutive spaces for 6 integers
- How much space is allocated?
- Heads up! The next bullets are a Big Deal.

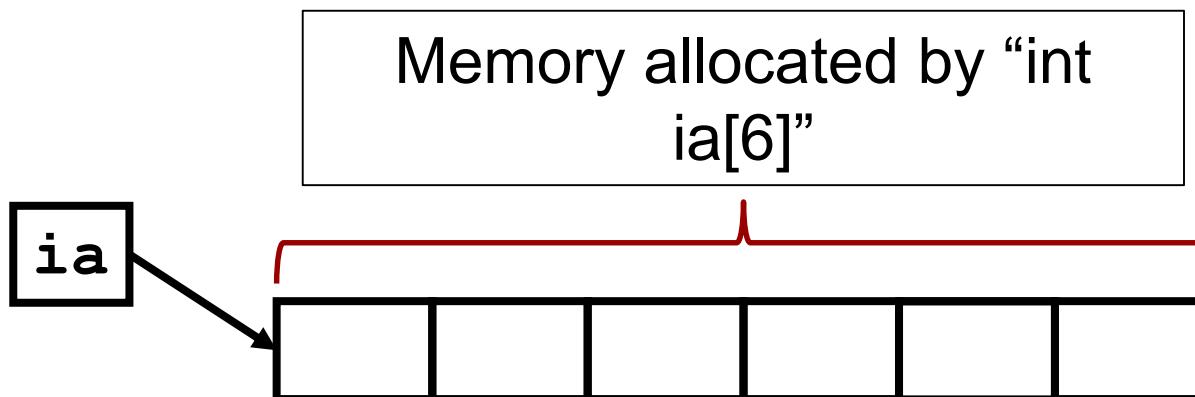


```
int ia[6];
```

- Allocates consecutive spaces for 6 integers
- How many bytes are allocated?

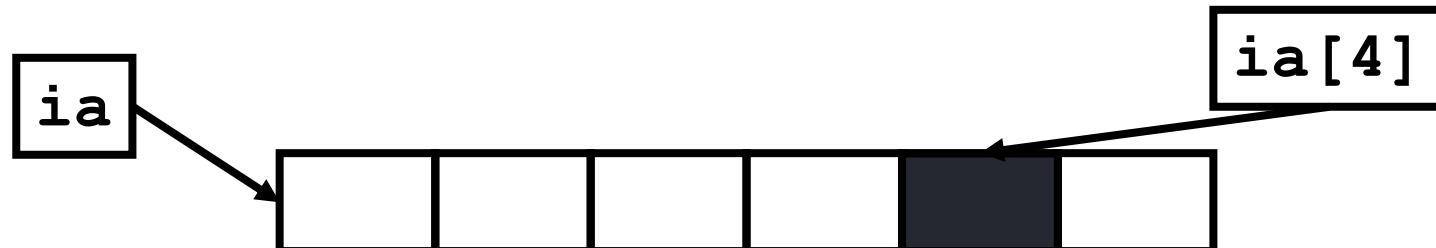
```
6 * sizeof(int) == 24
```

- It also creates the name **ia** when used in an expression is treated as a constant pointer to the first of the six integers



```
int ia[6];
```

- ↗ What does **ia[4]** mean? (In which context?)
- ↗ In an expression: **ia[4]** means ***exactly the same*** as **\*(ia + 4)**
- ↗ Multiply 4 by **sizeof(ia[0])**, add it to the value of **ia** and dereference and you address the fifth element of **ia**!  
 **$4 * 4 + \text{address of the array}$**
- ↗ By the way, what is **sizeof(ia)**?



```
int ia[6];
```

☞ Note:        **ia** ≡ **&ia[0]**

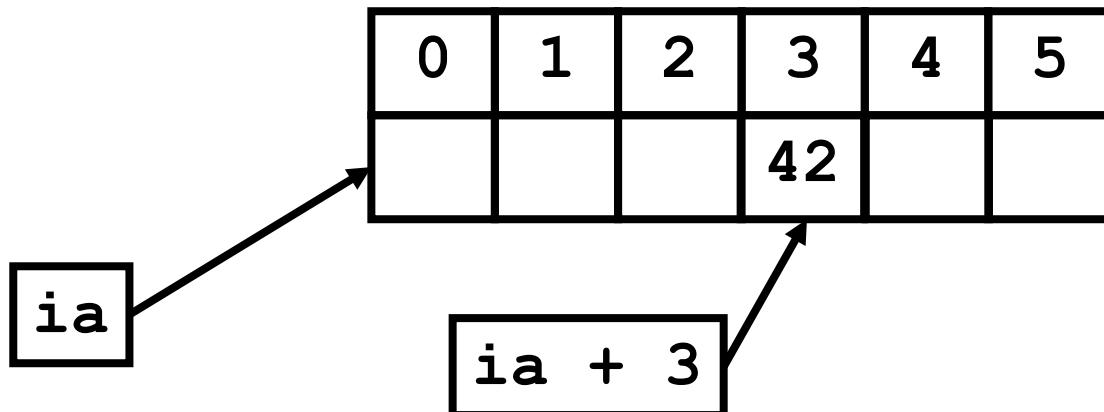
☞ Never say, "Pointers and arrays are exactly the same thing!!!" But they're really close!

```
int *ip;

ip = ia; /* Okay. */

ia = ip; /* Illegal. Why? */
```

```
int ia[6];
```



```
ia[3] = 42;
```

Using pointer arithmetic

```
* (ia + 3) = 42;
```

```
int ia[6];
```

```
ia[3] = 42;
```

↗ is the same as

```
* (ia + 3) = 42;
```

↗ and since addition is commutative

```
* (3 + ia) = 42;
```

↗ would imply that

```
3[ia] = 42;
```

should work.

↗ Does it? Is it a good idea?

1. Works, bad idea
2. Works, good idea
3. Doesn't work

```
#define MAX 6
int ia[MAX];
int *ip;
ip = ia; /* #1 */ Which one is illegal?
ia = ip; /* #2 */
ip[2] = 87; /* #3 */
```

\*ip      Points to what?

ip++;      Adds how much to IP?

\*ip      Now refers to what?

```
#define MAX 6
int ia[MAX];
int *ip;
ip = ia; /* #1 */
ia = ip; /* #2 */
ip[2] = 87; /* #3 */
```

Which one is illegal?

\*ip      Points to what?

ia[0]

ip++;      Adds how much to IP?

sizeof(int)

\*ip      Now refers to what?

ia[1]

# Question

You have the definitions

```
double fpa[100];
double *fpp;
```

Which of these statements is not legal?

- A. `fpp = fpa;`
- B. `fpp = &fpa[3];`
- C. `fpa = fpp + 3;`
- D. `fpa[2] = *(fpp + 3);`



# More pointer arithmetic

```
int i;
int ia[MAX];
for(i = 0; i < MAX; i++)
 ia[i] = 0;
```

```
int *ip;
int ia[MAX];
for(ip = ia; ip < ia + MAX; ip++)
 *ip = 0;
```

Sometimes pointer arithmetic is faster than  
subscripting, but these days the optimizer will unroll  
the multiplication, so it rarely matters!

# More Fun with Swap?

```
int arr1[] = {1, 2, 3};
int arr2[] = {9, 8, 7, 6};
int *p1 = arr1;
int *p2 = arr2;
/* What function call is needed here? */
for(i=0; i<4; i++) {
 printf("%d ", p1[i]);
}
for(i=0; i<3; i++) {
 printf("%d ", p2[i]);
}
```

**9 8 7 6 1 2 3**

# More Fun with Swap?

```
int arr1[] = {1, 2, 3};
int arr2[] = {9, 8, 7, 6};
int *p1 = arr1;
int *p2 = arr2;
swap_pointers(&p1, &p2);
for(i=0; i<4; i++) {
 printf("%d ", p1[i]);
}
for(i=0; i<3; i++) {
 printf("%d ", p2[i]);
}
```

**9 8 7 6 1 2 3**

# How Do We Swap Pointers?

```
void swap_pointers(int *a, int *b) {
 /* Choice 1 */
 int t;
 t = *a;
 *a = *b;
 *b = t;
}
void swap_pointers(int **a, int **b) {
 /* Choice 2 */
 int *t;
 t = *a;
 *a = *b;
 *b = t;
}
void swap_pointers(int **a, int **b) {
 /* Choice 3 */
 int **t;
 *t = *a;
 *a = *b;
 *b = *t;
}
```



# Introduction to GBA



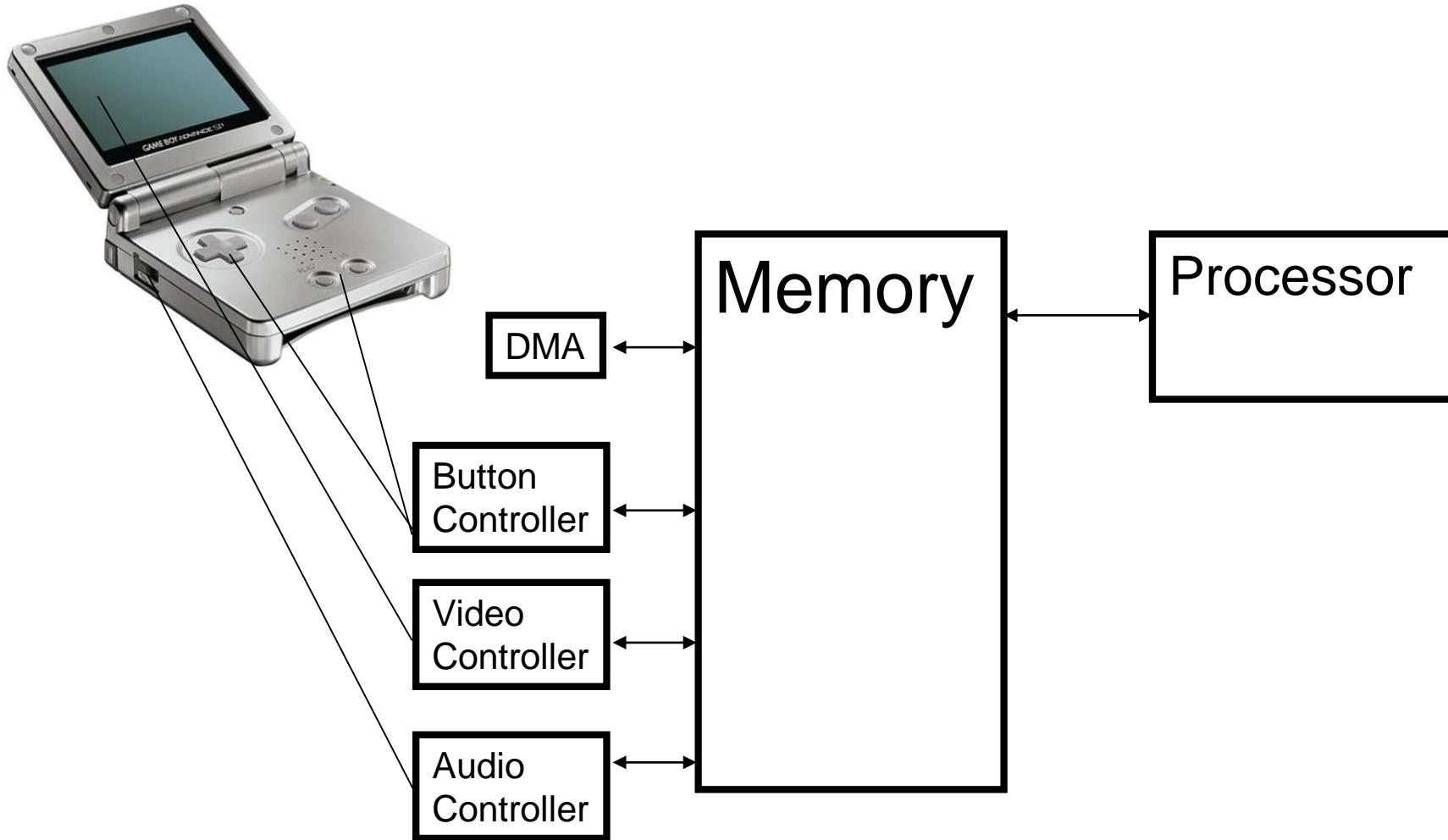
- ↗ Why GBA?
- ↗ Von Neumann Components
  - ↗ Processor/Control
  - ↗ Memory
  - ↗ Input
  - ↗ Output
- ↗ Other Goodies
- ↗ Programming the GBA

# Why GBA?

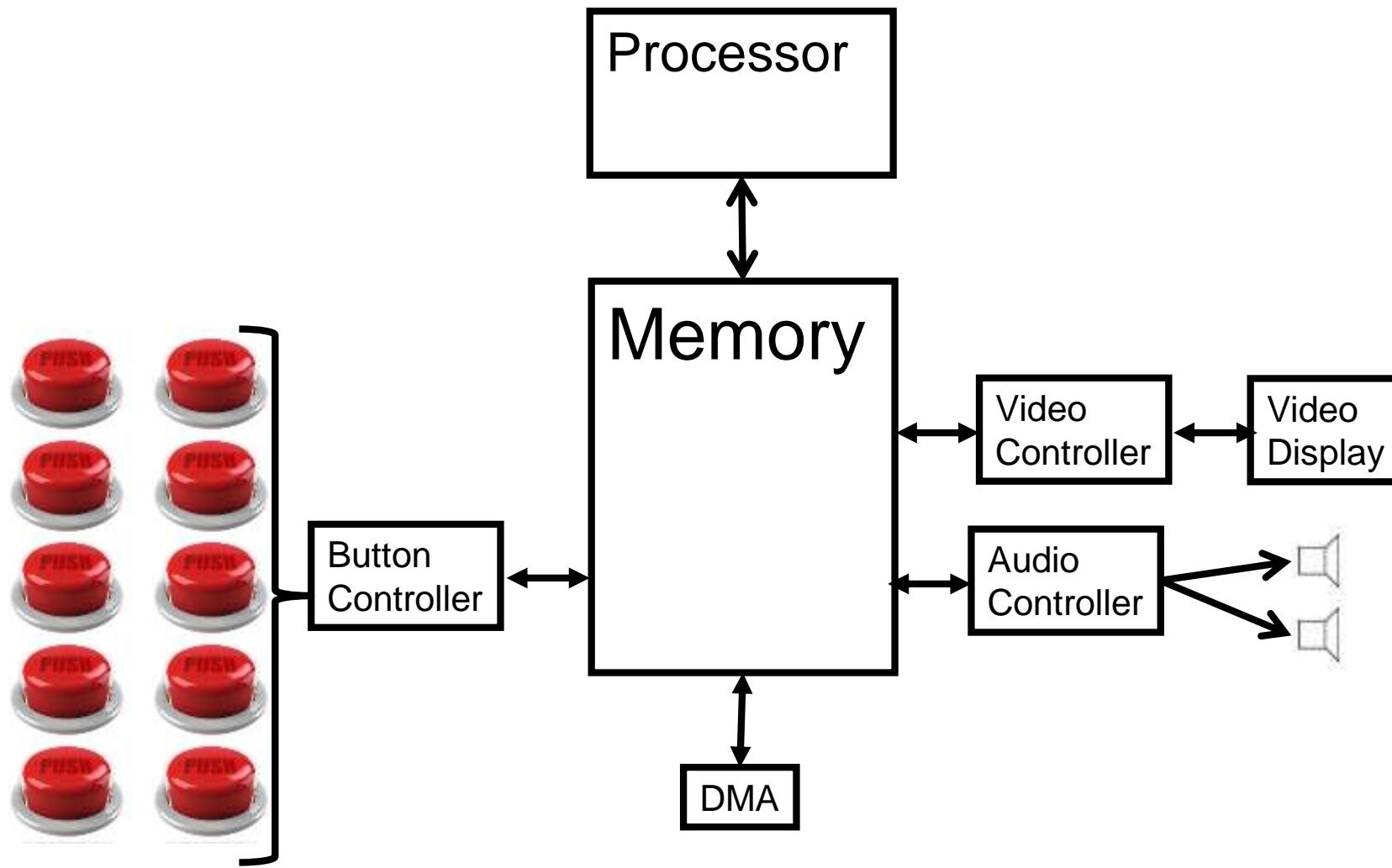
- ↗ Programming on bare metal
- ↗ No operating system
- ↗ Makes C seem more obvious
- ↗ Relatively simple hardware
- ↗ Slow...understand performance tradeoffs
- ↗ Fun!



# Functional Diagram



# Functional Diagram



- ARM-7-TDMI 3 stage pipelined RISC processor
- Very popular embedded processor in 2000-2001
- 16.78 MHz clock speed
- Datatypes
  - 8 bit (char)
  - 16 bit (short)
  - 32 bit (int)
- 32 bit address space

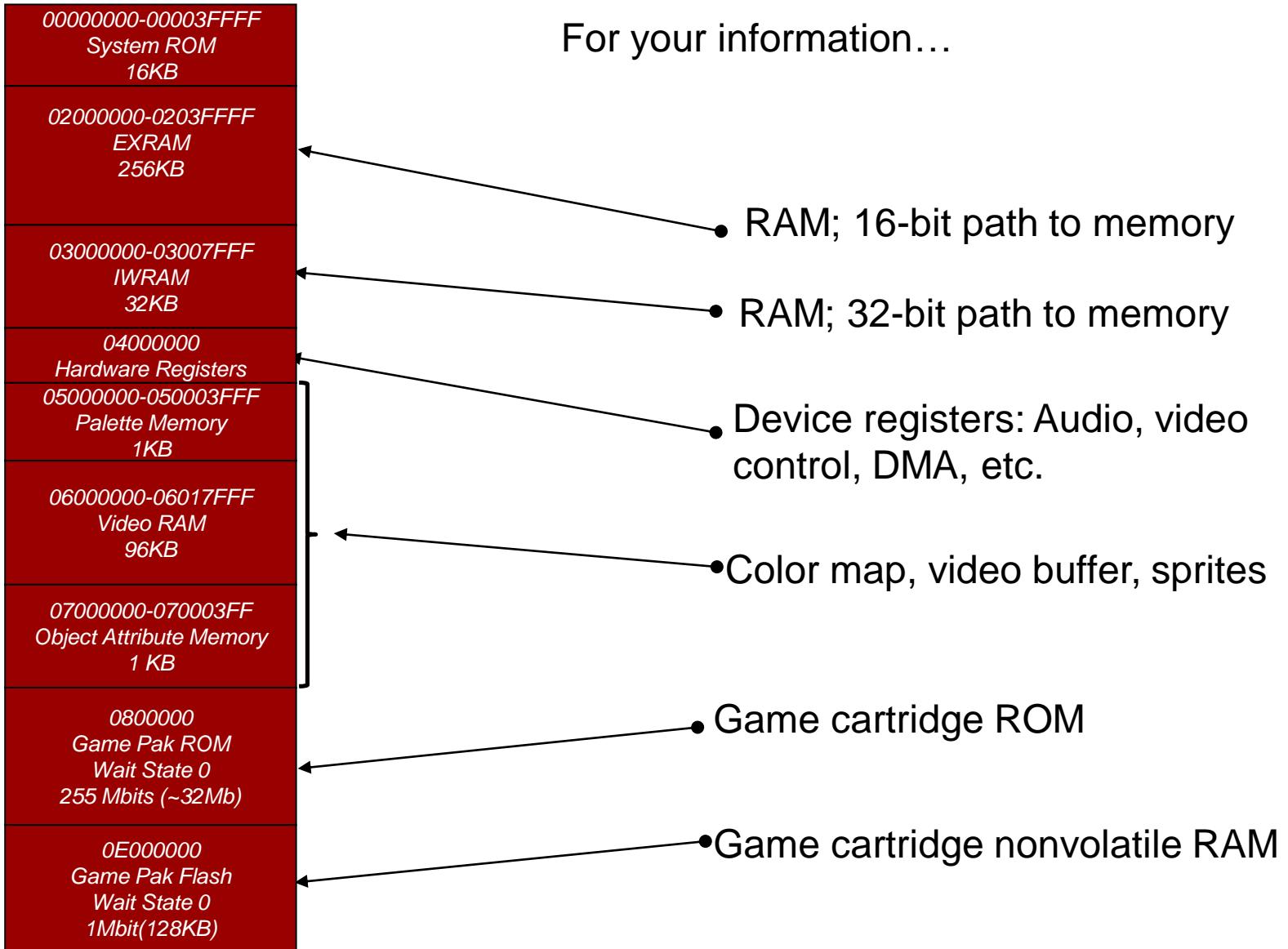
# ARM 7 Processor Apps

- ↗ Audio processing in the SEGA Dreamcast
- ↗ D-Link DSL-604+ Wireless ADSL Router[2]
- ↗ iPod from Apple
- ↗ iriver portable digital audio players (the H10 uses a chip with this processor)
- ↗ Juice Box
- ↗ Lego Mindstorms NXT
- ↗ Most of Nokia's mobile phone range.
- ↗ Nintendo DS (co-processor) and Game Boy Advance from Nintendo
- ↗ PocketStation
- ↗ Roomba 500 series from iRobot
- ↗ Sirius Satellite Radio receivers
- ↗ The main CPU in Stern Pinball S.A.M System games.
- ↗ Many automobiles embed ARM7 cores.

- $0x1 = 1$
- $0x10 = 16$
- $0x100 = 256$
- $0x1000 = 4,096$
- $0x1\ 0000 = 65,536$
- $0x10\ 0000 = 1,048,576$  (1 M)
- $0x100\ 0000 = 16,777,216$  (16 M)
- $0x1000\ 0000 = 268,435,456$  (256 M)
- $0x1\ 0000\ 0000 = 4,294,967,296$  (4 G)

# Typical GBA Memory Map

For your information...



# Question

How big is a *short* on the GBA?

- A. 8 bits
- B. 16 bits
- C. 32 bits
- D. 64 bits



Today's number is 16,024

- 10 buttons

- Start
- Select
- A
- B
- Left
- Right
- Up
- Down
- Left shoulder
- Right shoulder

- One button register
- 1 bit per button
  - 0 pressed
  - 1 not pressed

# Output

- ↗ 240 x 160 pixel color video display screen
- ↗ 6 display modes
  - ↗ Bit mapped
  - ↗ Tiled
- ↗ Hardware support for sprites



# Output

- ↗ Sound effect generators
- ↗ Direct Sound Hardware (DAC)

# Other Goodies

- ↗ Interrupts
- ↗ Timers
- ↗ DMA

- ↗ Download and install the software from Canvas (part of the Docker container you already have)
- ↗ Download tutorials
  - ↗ <http://www.coranac.com/tonc/text/>
  - ↗ (Mode 3 and DMA, sections 5 and 14)
- ↗ etc.

# Light Up a Pixel



## ↗ Integers

- ↗ (All can be signed, the default, or unsigned)
- ↗ char (1 byte)
- ↗ short int OR short (2 bytes)
- ↗ int (4 bytes)
- ↗ long int OR long (8 bytes)

## ↗ Floating Point

- ↗ (To be avoided because they're software emulated)
- ↗ Float
- ↗ double

# Old Friends?

- ↗ Remember our friends, the **device registers**?
- ↗ GBA has quite a few
- ↗ One register controls the many video modes on the GBA
- ↗ REG\_DISPCTL is at 0x0400 0000
- ↗ And by the way, how do we touch that device register in C?
  - ↗ `(* (unsigned short *) 0x04000000)`
  - ↗ or better yet
    - #define REG\_DISPCTL (\* (unsigned short \*) 0x04000000)
- ↗ We'll be using Mode 3 and BG2 for our programming!

# REG\_DISPCTL



- ↗ Bits 0-2 Mode
  - ↗ 0,1,2 Tile Modes
  - ↗ 3, 4, 5 Bitmap Modes
- ↗ For bitmapped graphics use BG2

# REG\_DISPCTL

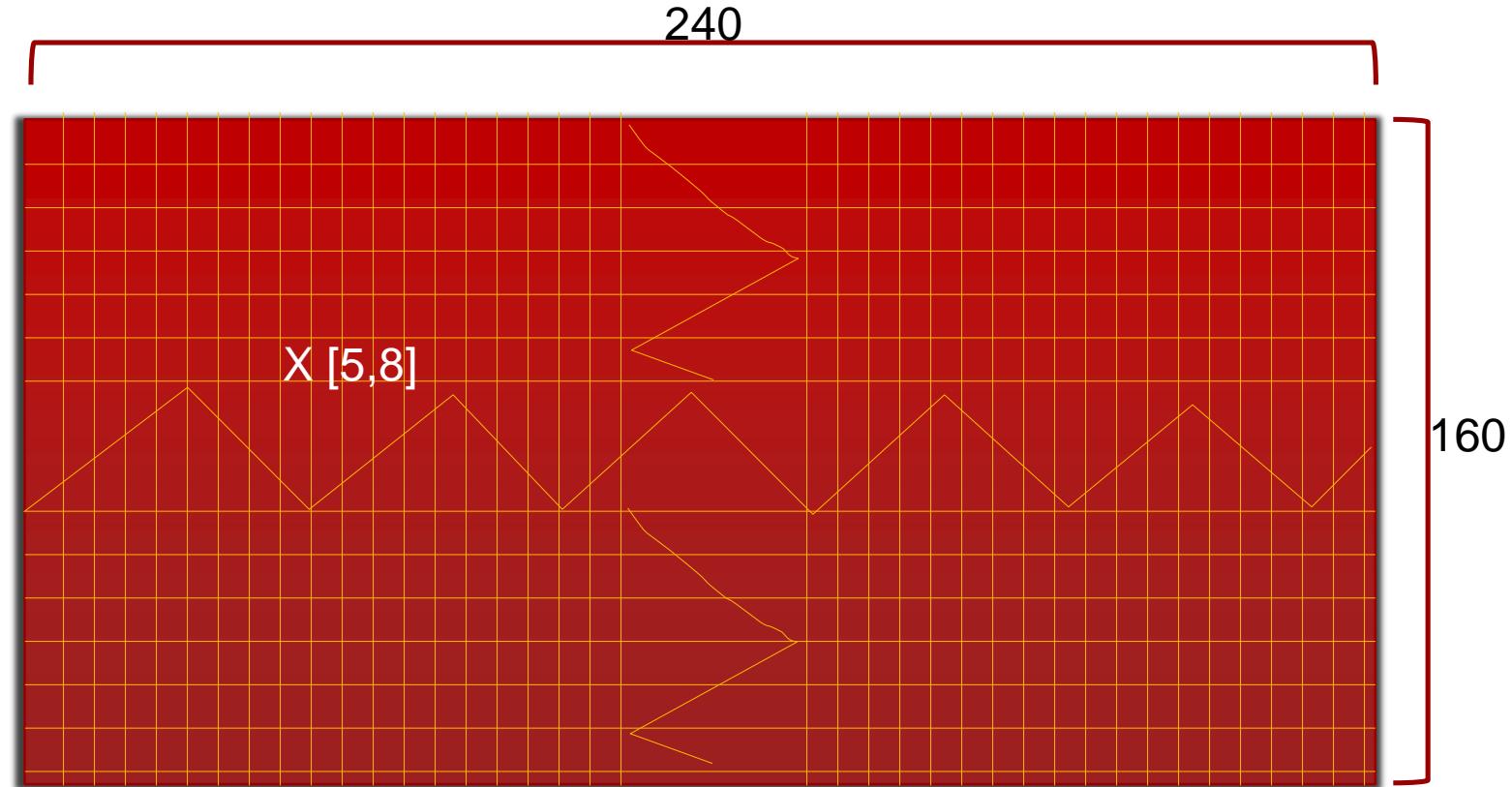
|                 |    |    |    |      |    |    |    |    |    |    |    |    |    |    |    |
|-----------------|----|----|----|------|----|----|----|----|----|----|----|----|----|----|----|
| 0F              | 0E | 0D | 0C | 0B   | 0A | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| 15              | 14 | 13 | 12 | 11   | 10 | 09 | 08 | 07 | 06 | 05 | 04 | 03 | 02 | 01 | 00 |
| BG3 BG2 BG1 BG0 |    |    |    | Mode |    |    |    |    |    |    |    |    |    |    |    |

- ↗ Bits 0-2 Mode
  - ↗ 0,1,2 Tile Modes
  - ↗ 3, 4, 5 Bitmap Modes
- ↗ For bitmapped graphics use BG2
- ↗  $100\ 0000\ 0011_2 = 0x403$
- ↗  $10000000011_2 = 2^{10} + 2^1 + 2^0 = 1024 + 2 + 1 = 1027$
- ↗ `REG_DISPCTL = 0x403;`

# Video Memory (in Mode 3)

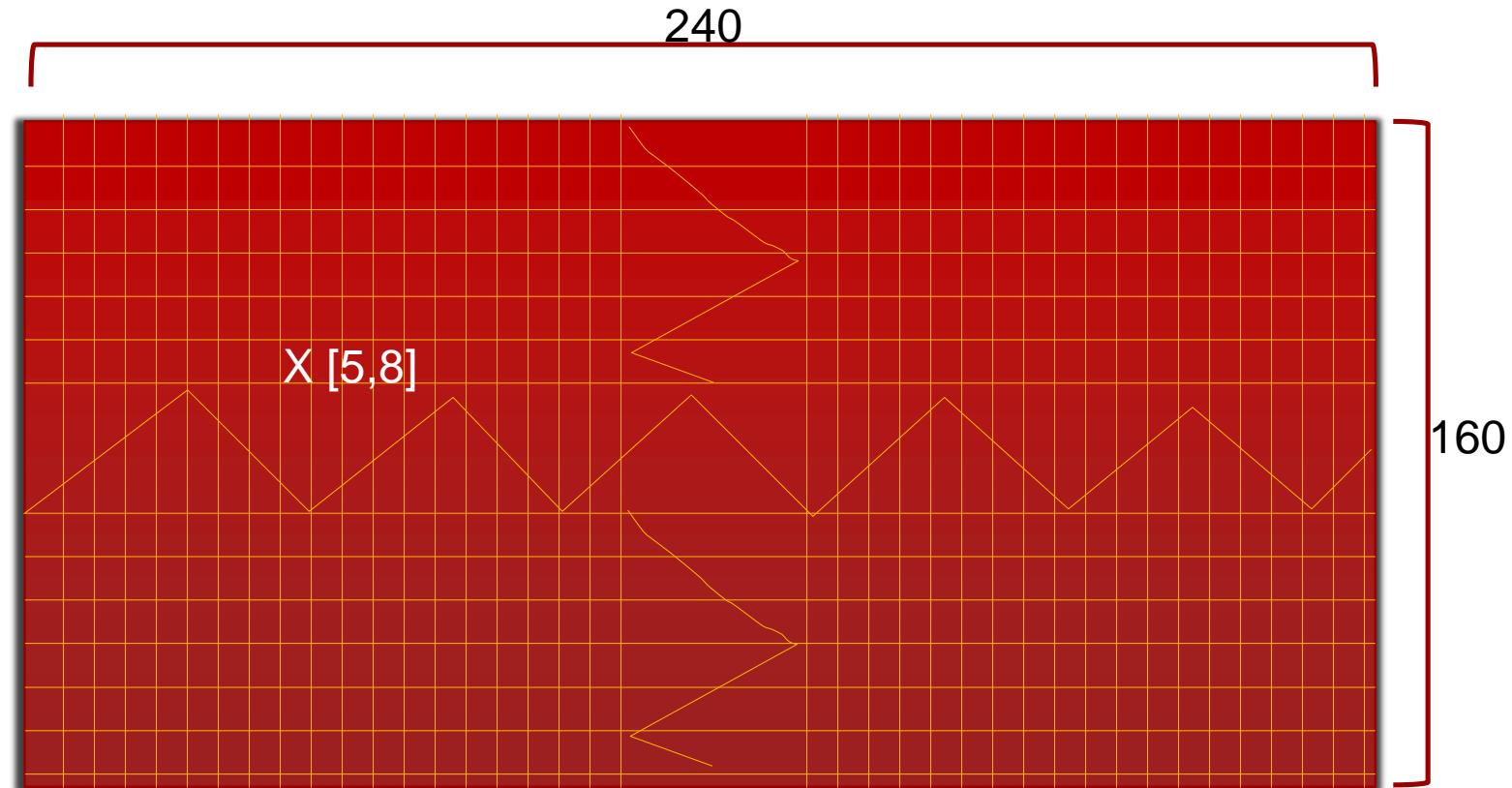
- Starts at 0x0600 0000
- Consists of 160 x 240 16-bit unsigned shorts
- Rows are assigned contiguously, one after the next
- How can we access this memory from C, starting at this address?
- What is the address of the pixel at
  - Row 5
  - Column 8

# Video Memory



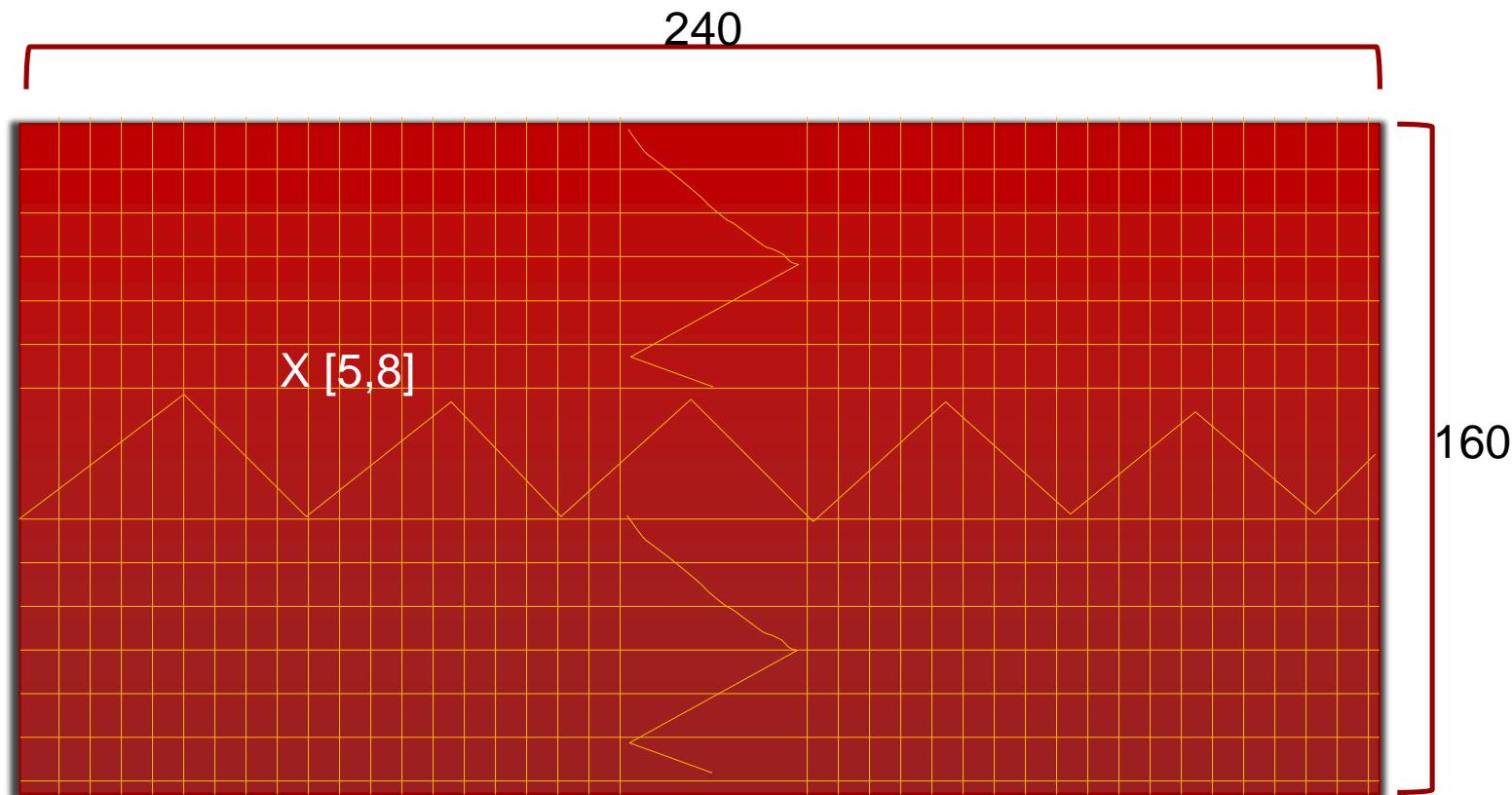
- ↗ How do we get to the pixel at row 5, col 8?
- ↗ What does this question remind you of?

# Video Memory



- ↗ Looks like a two-dimensional array stored in row major order, right?
- ↗ So how do we calculate an offset from the beginning?

# Video Memory

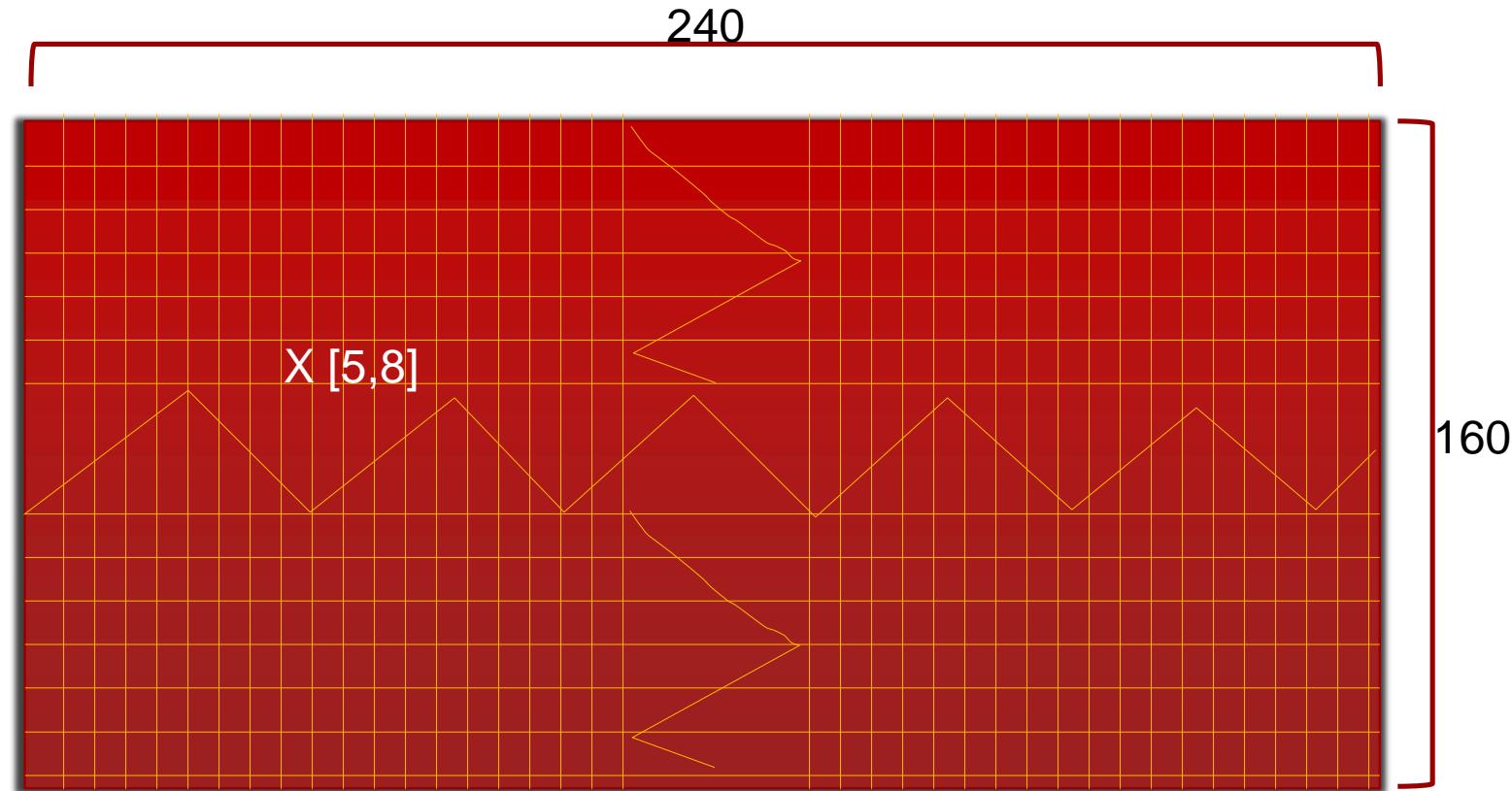


↗ How about  $\text{row} * \text{cols} + \text{col}$ , i.e.

$$\text{offset} = 5 * 240 + 8$$

$$\text{offset} = 1208$$

# Video Memory



- Now that we know the offset into the array, just add the base

```
offset = 5 * 240 + 8;
```

```
* ((unsigned short *) 0x6000000 + offset)
```

# And What Does a Pixel Look Like?

- Video Memory (in Mode 3, which is all we'll talk about)

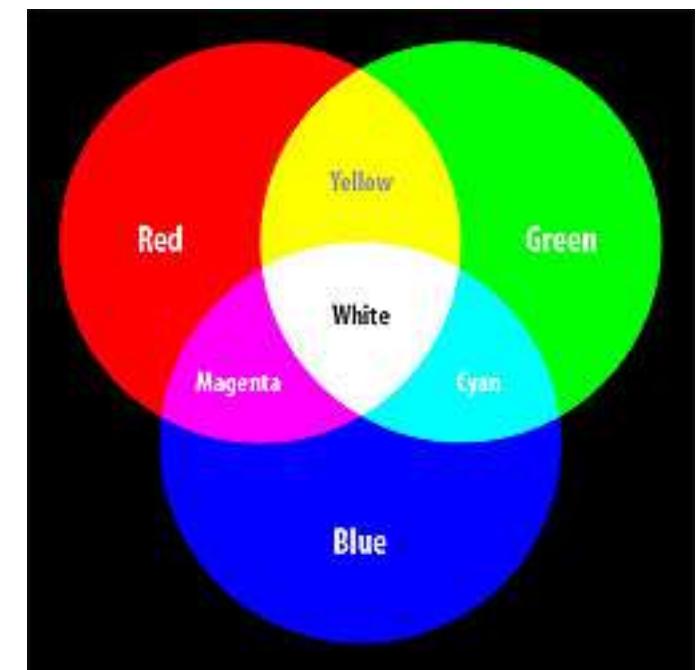
- How do we control the color?



- What would white be?

- In Hex
- In Decimal

- $0x7ff = 32767$



# Question

How would we calculate the offset of the pixel at row 29, column 80?

A.  $\text{offset} = 29 * 240 + 80$



B.  $\text{offset} = 80 * 160 + 29$

C.  $\text{offset} = 80 * 240 + 29$

D.  $\text{offset} = 29 * 160 + 80$

# Question

What color does 0x3E0 represent on the GBA?

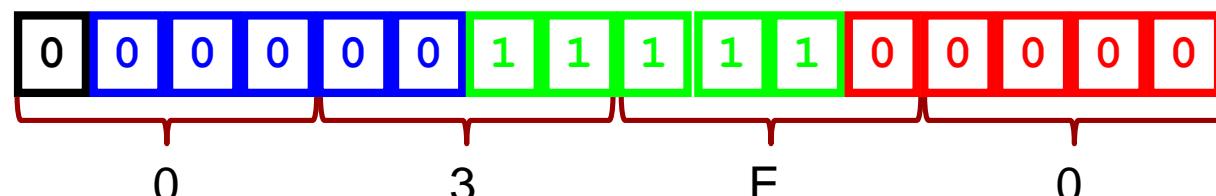
A. Red

B. Blue

C. Green ←

D. Yellow

E. Magenta



# So How Do We Set That Pixel?

- We know the base of video memory is 0x0600000, so

```
unsigned short *videoBuffer =
 (unsigned short *) 0x0600000;
videoBuffer[5*240+8] = 0x7fff;
```

- Could we also say

```
* (videoBuffer + 5*240+8) = 32767;
```

# How Might We Do This in LC-3 Assembly?

- ↗ We have to **make the assumption that LC-3 has 32-bit memory** and registers, but byte addressing like the ARM

```
; unsigned short *videoBuffer =
; (unsigned short *) 0x06000000;
LD R0,VBADDR
ST R0,videoBuffer

; videoBuffer[5*240+8] = 0x7fff;
LD R0,videoBuffer
LD R1,offset
ADD R1,R1,R1 ; ; because a short is 2 bytes
ADD R0,R0,R1 ; ; pixel address is now in R0
LD R1,WHITE
STR R1,R0,#0
HALT

WHITE .fill 0x7fff
VBADDR .fill 0x06000000
videoBuffer .fill 0
offset .fill 1208 ; 5*240+8
```

# What About ARM Assembly?

```
@ unsigned short *videoBuffer = 0x6000000;

videoBuffer: .word 0x6000000
...
@ videoBuffer[5*240+8] = 0x7fff;

 ldr r1, .L3 @ Load addr of VideoBuffer
 ldr r3, [r3] @ Load contents (like LDI!)
 ldr r2, #2416 @ r2 = 2416
 add r3, r3, r2 @ r3 = r3 + r2
 ldr r2, .L4 @ load 0x7fff into r2
 strh r2, [r3] @ store halfword [like STI!]
...
 .align 2
.L3: .word videoBuffer
.L4: .word 0x7fff
```

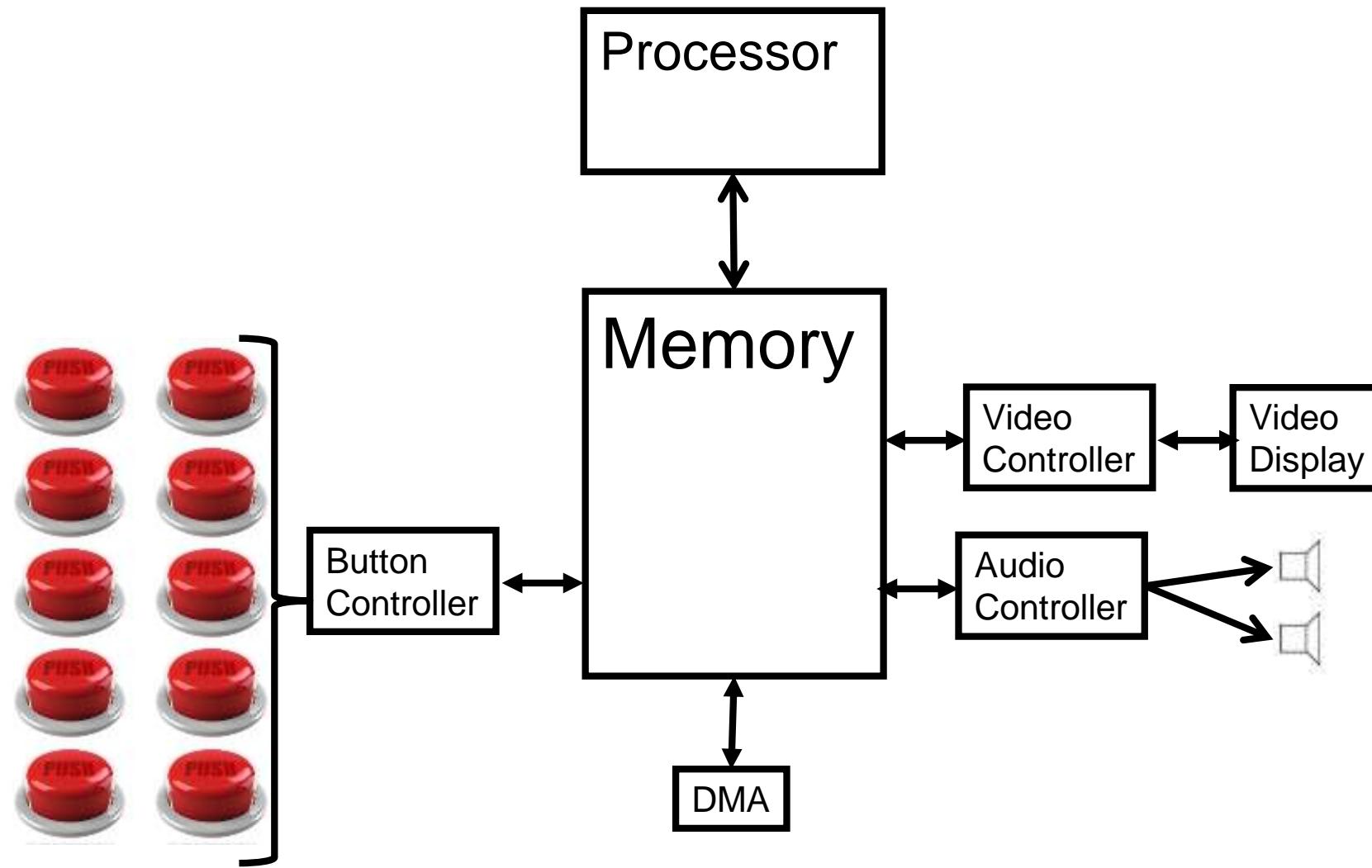
# GBA C Programming



# Quick Overview



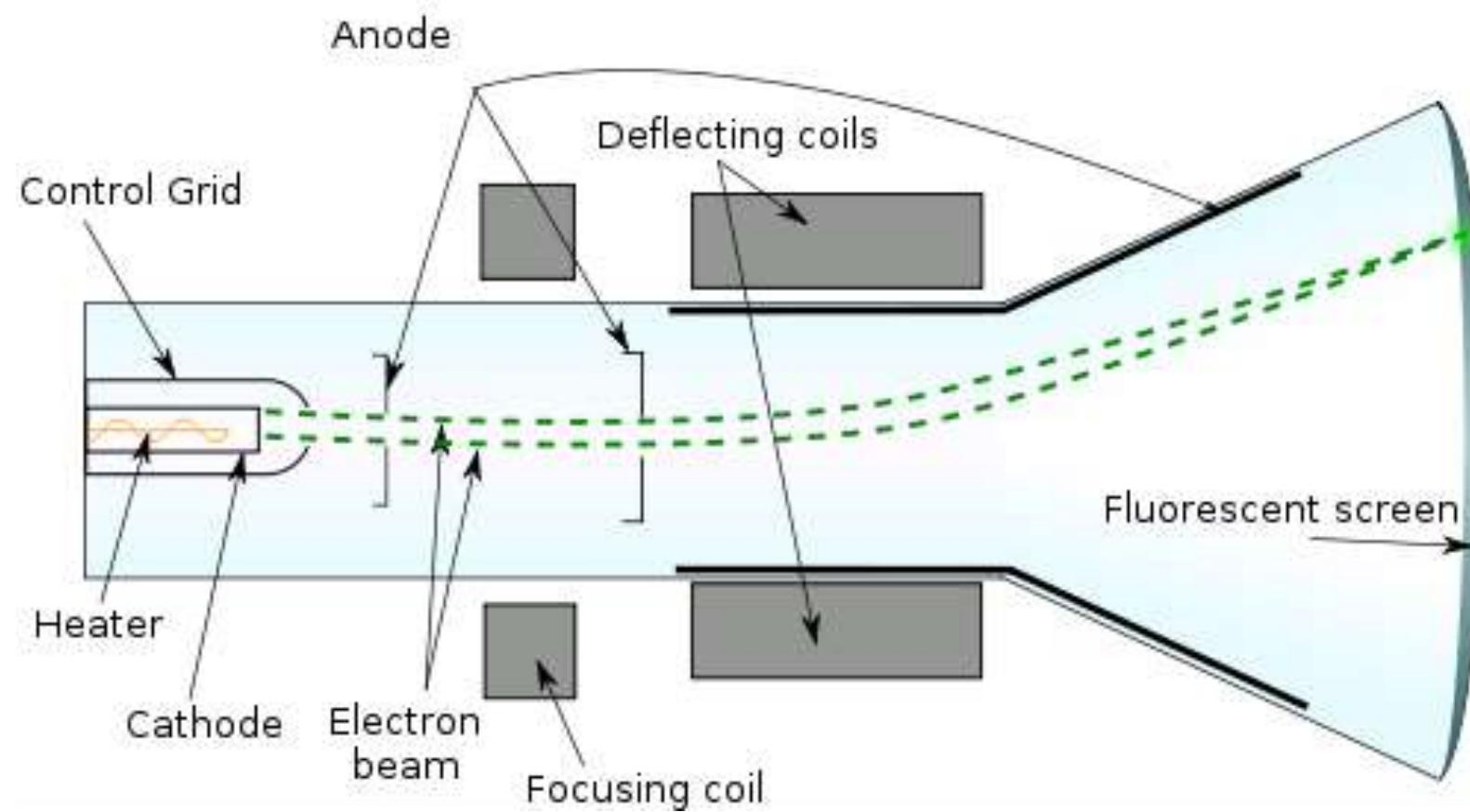
# Functional Diagram



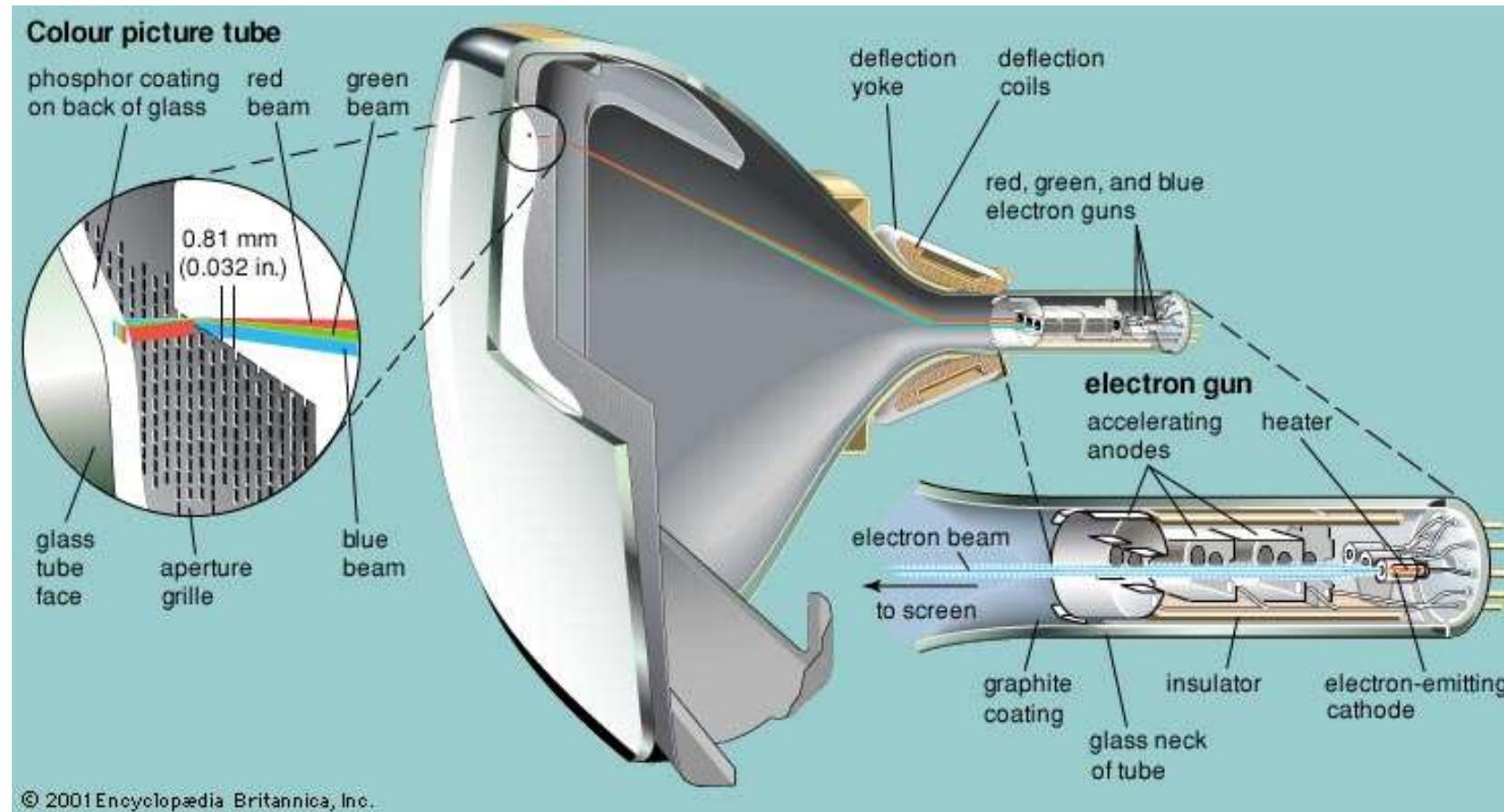
# Blanking Intervals



# Cathode Ray Tube

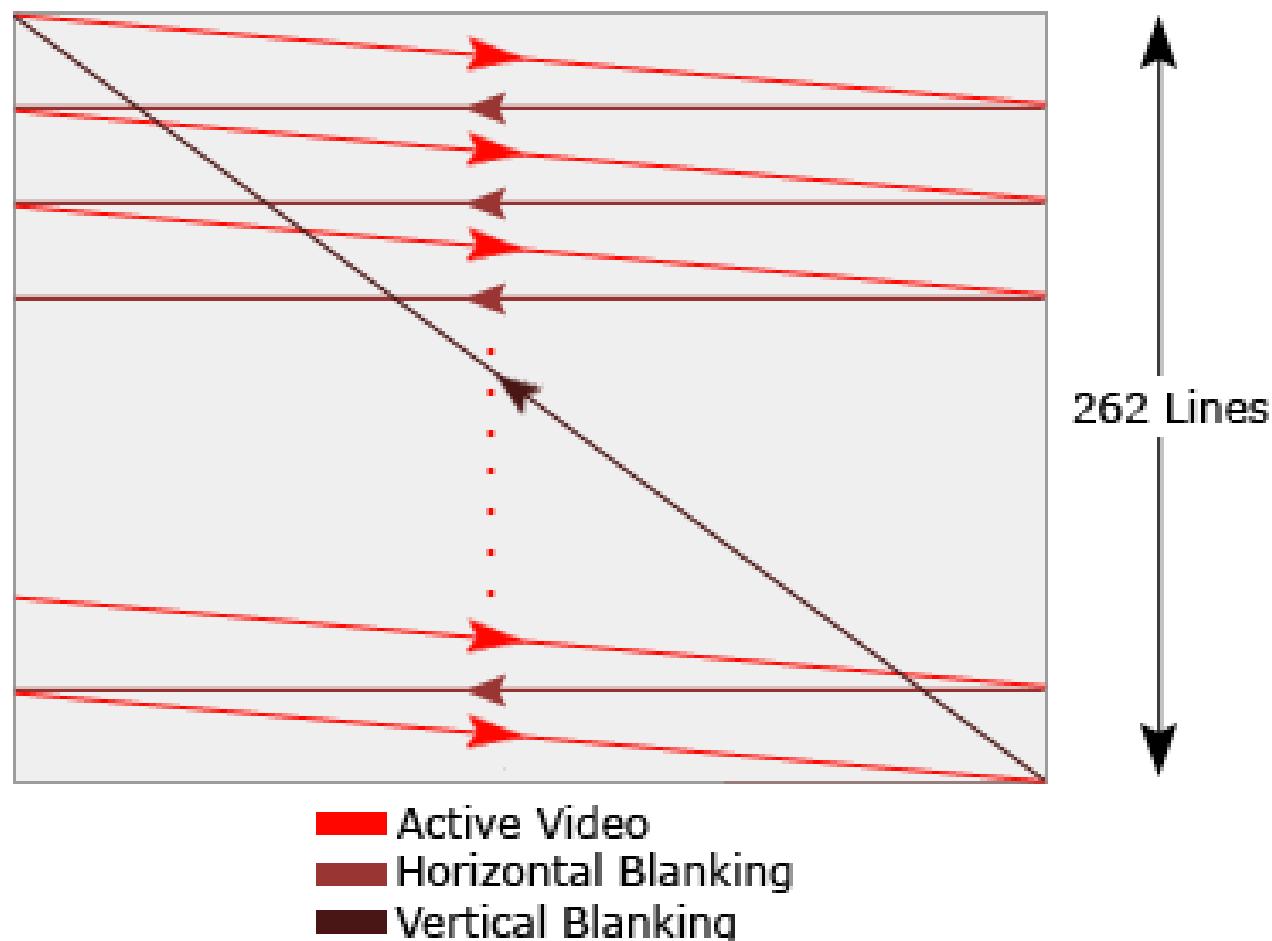


# Color CRT



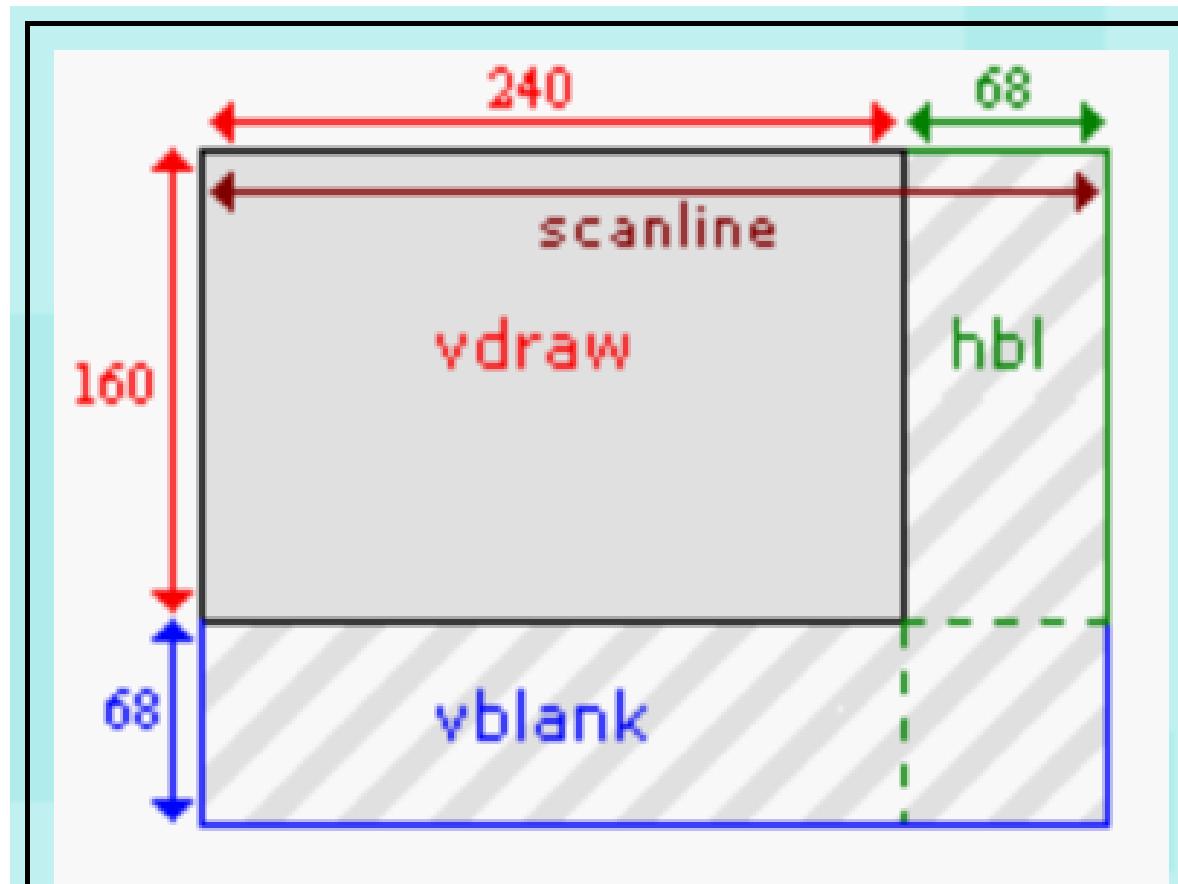
Note that in real life, the beams are **not** colored. Red, blue and green electrons are a myth. It's the phosphors on the screen that glow in colors!

# NTSC Scanning Pattern (interleaved)



- GBA screen is refreshed at 60 Hz
- There are pauses in the drawing process
  - For each scanline (160 lines)
    - Draw scanline (240 pixels)
    - Hblank (68 pixels)
  - Vblank (68 scanlines)
- To avoid tearing, positional data is usually updated at the VBlank. This is why most games run at 60 or 30 fps.
- FYI, this mimics the NTSC/PAL analog TV standards which required blanking intervals to reposition the electron beam.

# HBlank and VBlank



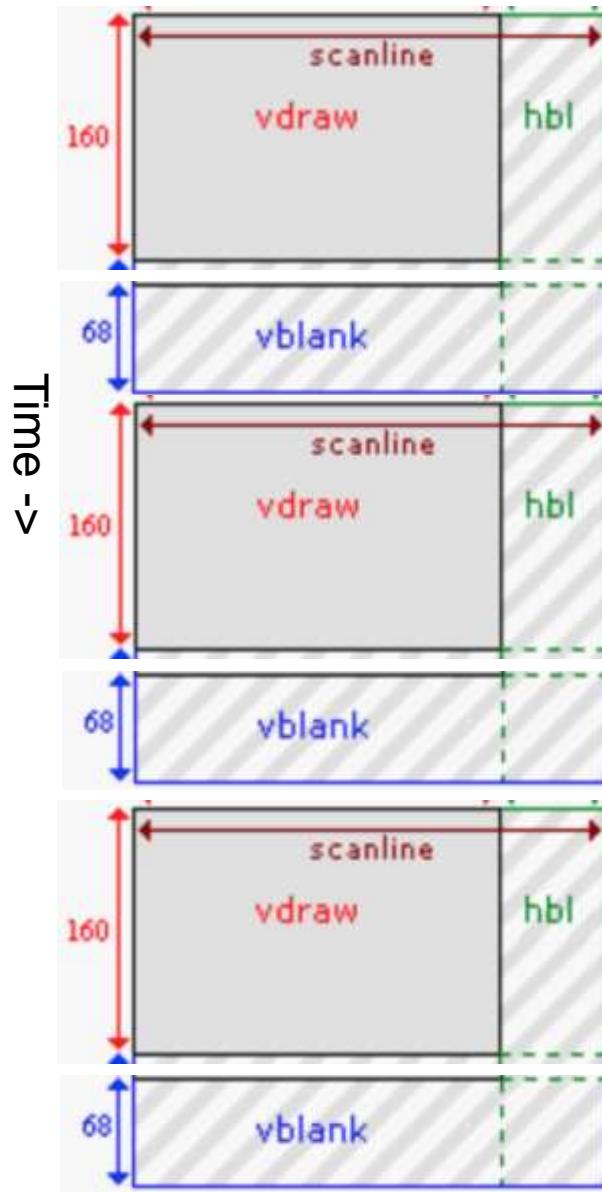
**Fig 4.1:** vdraw, vblank and hblank periods.

## Drawing Times

- A full screen refresh takes exactly 280896 cycles, divided by the clock speed gives a frame rate of 59.73.
- From the Draw/Blank periods given above you can see that there are 4 cycles per pixel, and 1232 cycles per scanline.

| subject  | length                  | cycles |
|----------|-------------------------|--------|
| pixel    | 1                       | 4      |
| HDraw    | 240px                   | 960    |
| HBlank   | 68px                    | 272    |
| scanline | Hdraw+Hbl               | 1232   |
| VDraw    | $160 * \text{scanline}$ | 197120 |
| VBlank   | $68 * \text{scanline}$  | 83776  |
| refresh  | VDraw+Vbl               | 280896 |

**Table 4.1:** Display timing details



**2 during vdraw:** perform other useful computations

**1 during vblank:** make changes to the videoBuffer



**2 during vdraw:** perform other useful computations

**1 during vblank:** make changes to the videoBuffer



**2 during vdraw:** perform other useful computations

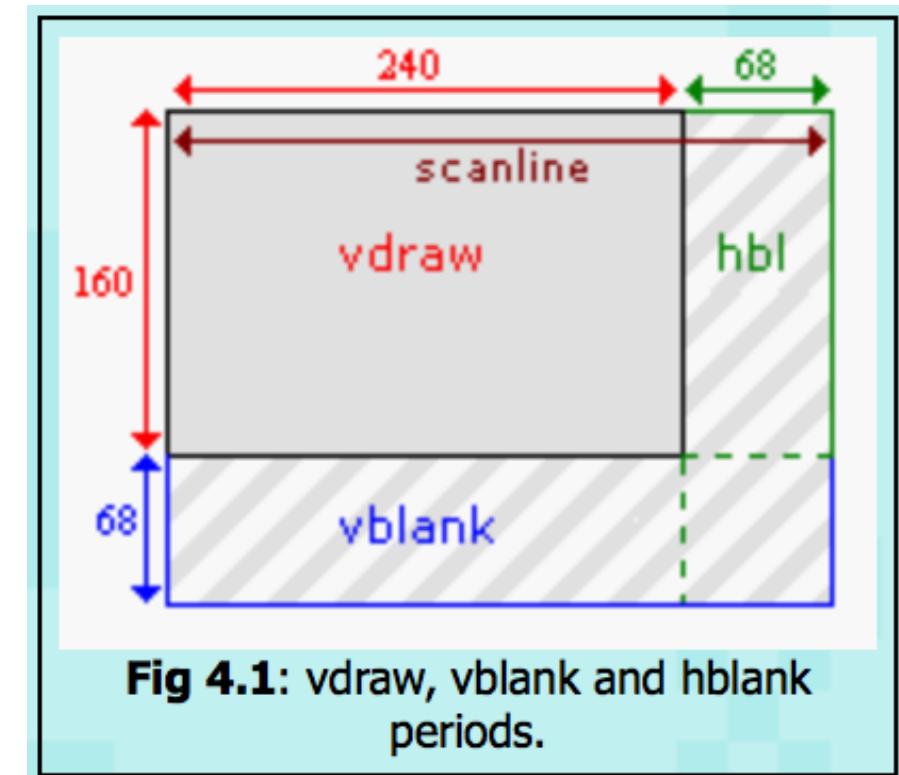
**1 during vblank:** make changes to the videoBuffer

# Question

How many scan line times are available during the vertical blanking interval on the GBA?

- A. 60
- B. 64
- C. 68 ←
- D. 72
- E. 160

Today's number is 68,068



# Button Input



- 10 buttons (bit 0 through bit 9 of button register)s
  - A
  - B
  - Start
  - Select
  - Right
  - Left
  - Up
  - Down
  - Right shoulder
  - Left shoulder

- One button register
- 1 bit per button
  - 0 pressed
  - 1 not pressed

# Definitions

```
#define BUTTONS * (volatile
 unsigned int *) 0x4000130
```

```
#define KEY_DOWN_NOW(key)
 (~ (BUTTONS) & key)
```

# Question

The positions of the GBA's ten buttons can be discovered programmatically by

- A. Executing a special I/O instruction
- B. Reading from a device register
- C. Reading a word from memory
- D. Loading the button register into a general purpose register



DMA



# What is DMA?

- DMA = Direct Memory Access
- Hardware supported data copy
  - Up to 10x as fast as array copies
  - You set it up, the CPU is halted, data is transferred, and CPU gains back control
    - Careful—reckless use can block interrupts

- Channel 0
  - Highest Priority
  - Time Critical Operations
  - Only works with IWRAM
- Channel 1 & 2
  - Transfer sound chunks to sound buffer
- Channel 3
  - Lowest Priority
  - General purpose copies, like loading tiles or bitmaps into memory

- ↗ Source

- ↗ REG\_DM<sub>A</sub>xSAD (x = 0, 1, 2, 3)
- ↗ The location of the data that will be copied

- ↗ Destination

- ↗ REG\_DM<sub>A</sub>xDAD
- ↗ Where to copy the data to

- ↗ Amount

- ↗ REG\_DM<sub>A</sub>xCNT (DMA control)
- ↗ How much to copy

- Lower 16 bits contain amount to transfer
- Upper 16 bits contain other options
- Turn on a DMA channel
- When to perform the DMA
- How the copy source and destination behave
- How much to copy at a time
- Whether or not to throw an interrupt on completion
- Repeat or don't repeat on finish
- Can be treated as one 32 bit register, or two 16 bit registers

# REG\_DMAMxCNT

| bits  | name | define                                                       | description                                                                                                                                                                                                                                                                          |
|-------|------|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 0-15  | N    | Number of transfers.                                         |                                                                                                                                                                                                                                                                                      |
| 21-22 | DA   | DMA_DST_INC<br>DMA_DST_DEC<br>DMA_DST_FIXED<br>DMA_DST_RESET | 00: increment after each transfer<br>(default)<br>01: decrement after each transfer<br>10: none; address is fixed<br>11: haven't used it yet, but<br>apparently this will increment<br>the destination during the<br>transfer, and reset it to the<br>original value when it's done. |
| 23-24 | SA   | DMA_SRC_INC<br>DMA_SRC_DEC<br>DMA_SRC_FIXED                  | Source Adjustment. Works just like the two bits<br>for the destination. Note that there is no<br>DMA_SRC_RESET; code 3 for source is forbidden.                                                                                                                                      |
| 25    | R    | DMA_REPEAT                                                   | Repeats the copy at each VBlank or<br>HBlank if the DMA timing has been set to<br>those modes.                                                                                                                                                                                       |

# REG\_DMAtCNT

| bits  | name | define                                                      | description                                                                                                                                                                                                                                                                                                                                                     |
|-------|------|-------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 26    | CS   | DMA_16,<br>DMA_32                                           | Chunk Size.<br>Sets DMA to copy by halfword (if clear) or word (if set).                                                                                                                                                                                                                                                                                        |
| 28-29 | TM   | DMA_NOW<br>DMA_AT_VBLANK<br>DMA_AT_HBLANK<br>DMA_AT_REFRESH | 00: start immediately<br>01: start at VBlank<br>10: start at HBlank.<br>11: Never used it so far, but here's how I gather it works. For DMA1 and DMA2 it'll refill the FIFO when it has been emptied. Count and size are forced to 1 and 32bit, respectively. For DMA3 it will start the copy at the start of each rendering line, but with a 2 scanline delay. |
|       |      |                                                             | Timing Mode. Specifies when the transfer should start.                                                                                                                                                                                                                                                                                                          |
| 30    | I    | DMA_IRQ                                                     | Interrupt request. Raise an interrupt when finished.                                                                                                                                                                                                                                                                                                            |
| 31    | En   | DMA_ON                                                      | Enable the DMA transfer for this channel.                                                                                                                                                                                                                                                                                                                       |

# Source Adjustment

- ↗ REG\_DMAMOD bits 23-24
- ↗ Incrementing source (default:  $00_2$ ) causes DMA to behave as a memory copy
- ↗ Fixing source ( $10_2$ ) causes DMA to behave as a memory fill
  - ↗ Copy the same thing over and over to a stretch of memory
  - ↗ Fill the screen with a color, clear a tilemap to all zeros, etc
    - ↗ Careful! SAD takes an address, not a value!
    - ↗ ***Make local variables volatile if using their address for a DMA fill***

- We map a struct array over the DMA registers

```
typedef struct {
 const volatile void *src;
 void *dst;
 u32 cnt;
} DMA_CONTROLLER;

#define DMA((volatile DMA_CONTROLLER *)
 0x040000b0)
```

## DMA Setup (cont)

```
#define DMA_TRANSFER(_dst,_src,_count,_ch,_mode) \
do { \
 DMA[_ch].cnt = 0; \
 DMA[_ch].src = (const void*)(_src); \
 DMA[_ch].dst = (void*)(_dst); \
 DMA[_ch].cnt = (_count) | (_mode); \
} while(0) \
 \
void dma_memcpy(void *dst, const void *src, u16 count) \
{ \
 DMA[3].cnt = 0; // shut off previous transfer \
 DMA[3].src = src; \
 DMA[3].dst = dst; \
 DMA[3].cnt = count | DMA_32 | DMA_ON; \
}
```

- Clearing a DMA register before its scheduled copy occurs will stop it from ever happening
  - Careful when using delayed DMA
- Even immediate DMA has a 2 cycle delay
  - DMA calls in immediate succession could cancel the earlier one
  - 2 cycles is short enough that returning from the DMA setup function allows the copy to begin before other code executes

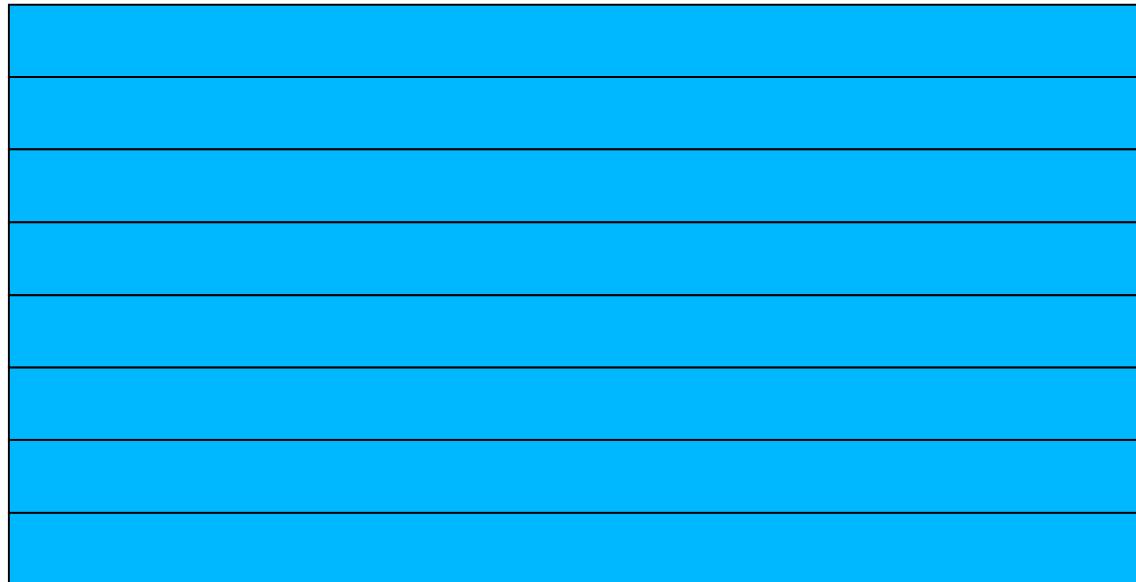
# Question

How many bytes are occupied by a GBA DMA channel's device registers?

- A. 6
- B. 8
- C. 12 
- D. 16
- E. 20

```
typedef struct {
 const volatile void *src; // 4
 void *dst; // 4
 u32 cnt; // 4
} DMA_CONTROLLER;
```

# Filling a Rectangle



# Why is DMA so *Fast*



# Questions?



# Loop Template

Initialize

While true

    Previous state = Current state

    Check buttons

    Calculate new values for Current state

    Wait for Vblank

    Undraw using Previous state

    Draw using Current state

# Loop Template

Set Gamestate to INITPLAY

While true

    Previous state = Current state

    if Gamestate == PLAY

        Check buttons

        Calculate new values for Current state

    Else if Gamestate == INITPLAY

        Initialize the Current state

Wait for Vblank

If Gamestate == PLAY

    Undraw using Previous state

    Draw using Current state

Else if Gamestate == INITPLAY

    Gamestate = PLAY

Text



# ASCII Table

| Decimal | Hex | Char                   | Decimal | Hex | Char    | Decimal | Hex | Char | Decimal | Hex | Char  |
|---------|-----|------------------------|---------|-----|---------|---------|-----|------|---------|-----|-------|
| 0       | 0   | [NULL]                 | 32      | 20  | [SPACE] | 64      | 40  | @    | 96      | 60  | `     |
| 1       | 1   | [START OF HEADING]     | 33      | 21  | !       | 65      | 41  | A    | 97      | 61  | a     |
| 2       | 2   | [START OF TEXT]        | 34      | 22  | "       | 66      | 42  | B    | 98      | 62  | b     |
| 3       | 3   | [END OF TEXT]          | 35      | 23  | #       | 67      | 43  | C    | 99      | 63  | c     |
| 4       | 4   | [END OF TRANSMISSION]  | 36      | 24  | \$      | 68      | 44  | D    | 100     | 64  | d     |
| 5       | 5   | [ENQUIRY]              | 37      | 25  | %       | 69      | 45  | E    | 101     | 65  | e     |
| 6       | 6   | [ACKNOWLEDGE]          | 38      | 26  | &       | 70      | 46  | F    | 102     | 66  | f     |
| 7       | 7   | [BELL]                 | 39      | 27  | '       | 71      | 47  | G    | 103     | 67  | g     |
| 8       | 8   | [BACKSPACE]            | 40      | 28  | (       | 72      | 48  | H    | 104     | 68  | h     |
| 9       | 9   | [HORIZONTAL TAB]       | 41      | 29  | )       | 73      | 49  | I    | 105     | 69  | i     |
| 10      | A   | [LINE FEED]            | 42      | 2A  | *       | 74      | 4A  | J    | 106     | 6A  | j     |
| 11      | B   | [VERTICAL TAB]         | 43      | 2B  | +       | 75      | 4B  | K    | 107     | 6B  | k     |
| 12      | C   | [FORM FEED]            | 44      | 2C  | ,       | 76      | 4C  | L    | 108     | 6C  | l     |
| 13      | D   | [CARRIAGE RETURN]      | 45      | 2D  | -       | 77      | 4D  | M    | 109     | 6D  | m     |
| 14      | E   | [SHIFT OUT]            | 46      | 2E  | .       | 78      | 4E  | N    | 110     | 6E  | n     |
| 15      | F   | [SHIFT IN]             | 47      | 2F  | /       | 79      | 4F  | O    | 111     | 6F  | o     |
| 16      | 10  | [DATA LINK ESCAPE]     | 48      | 30  | 0       | 80      | 50  | P    | 112     | 70  | p     |
| 17      | 11  | [DEVICE CONTROL 1]     | 49      | 31  | 1       | 81      | 51  | Q    | 113     | 71  | q     |
| 18      | 12  | [DEVICE CONTROL 2]     | 50      | 32  | 2       | 82      | 52  | R    | 114     | 72  | r     |
| 19      | 13  | [DEVICE CONTROL 3]     | 51      | 33  | 3       | 83      | 53  | S    | 115     | 73  | s     |
| 20      | 14  | [DEVICE CONTROL 4]     | 52      | 34  | 4       | 84      | 54  | T    | 116     | 74  | t     |
| 21      | 15  | [NEGATIVE ACKNOWLEDGE] | 53      | 35  | 5       | 85      | 55  | U    | 117     | 75  | u     |
| 22      | 16  | [SYNCHRONOUS IDLE]     | 54      | 36  | 6       | 86      | 56  | V    | 118     | 76  | v     |
| 23      | 17  | [END OF TRANS. BLOCK]  | 55      | 37  | 7       | 87      | 57  | W    | 119     | 77  | w     |
| 24      | 18  | [CANCEL]               | 56      | 38  | 8       | 88      | 58  | X    | 120     | 78  | x     |
| 25      | 19  | [END OF MEDIUM]        | 57      | 39  | 9       | 89      | 59  | Y    | 121     | 79  | y     |
| 26      | 1A  | [SUBSTITUTE]           | 58      | 3A  | :       | 90      | 5A  | Z    | 122     | 7A  | z     |
| 27      | 1B  | [ESCAPE]               | 59      | 3B  | ;       | 91      | 5B  | [    | 123     | 7B  | {     |
| 28      | 1C  | [FILE SEPARATOR]       | 60      | 3C  | <       | 92      | 5C  | \    | 124     | 7C  |       |
| 29      | 1D  | [GROUP SEPARATOR]      | 61      | 3D  | =       | 93      | 5D  | ]    | 125     | 7D  | }     |
| 30      | 1E  | [RECORD SEPARATOR]     | 62      | 3E  | >       | 94      | 5E  | ^    | 126     | 7E  | ~     |
| 31      | 1F  | [UNIT SEPARATOR]       | 63      | 3F  | ?       | 95      | 5F  | _    | 127     | 7F  | [DEL] |

# fontdata\_6x8[]

```
/* num: 72 */ /* num: 97 */ /* num: 112 */ /* num: 121 */
0,1,0,0,0,1, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0,
0,1,0,0,0,1, 0,0,0,0,0,0, 0,0,0,0,0,0, 0,0,0,0,0,0,
0,1,0,0,0,1, 0,0,1,1,1,0, 0,1,1,1,1,0, 0,1,0,0,1,0,
0,1,1,1,1,1, 0,0,0,0,0,1, 0,1,0,0,0,1, 0,1,0,0,1,0,
0,1,0,0,0,1, 0,0,1,1,1,1, 0,1,0,0,0,1, 0,1,0,0,1,0,
0,1,0,0,0,1, 0,1,0,0,0,1, 0,1,0,0,0,1, 0,0,1,1,1,0,
0,1,0,0,0,1, 0,0,1,1,1,1, 0,1,1,1,1,0, 0,0,0,1,0,0,
0,0,0,0,0,0, 0,0,0,0,0,0, 0,1,0,0,0,0, 0,1,1,0,0,0,
```

# Continuing with C



## ➤ C Arrays

- Arrays of pointers (marginally indexed arrays)
- Multidimensional (row major order)
- Memory Layout

# Arrays of Pointers



# Arrays of Pointers (or Marginally Indexed Arrays)

```
char *month_name(int n) {
 static char *name[] = {
 "Illegal month",
 "January", "February", "March",
 "April", "May", "June",
 "July", "August", "September",
 "October", "November", "December"
 } ;

 return (n<1 || n>12) ? name[0] : name[n] ;
}
```

# Arrays of Pointers

- A block of memory (probably in the constant area) is initialized like this for all the **string constants**:

```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

# Arrays of Pointers

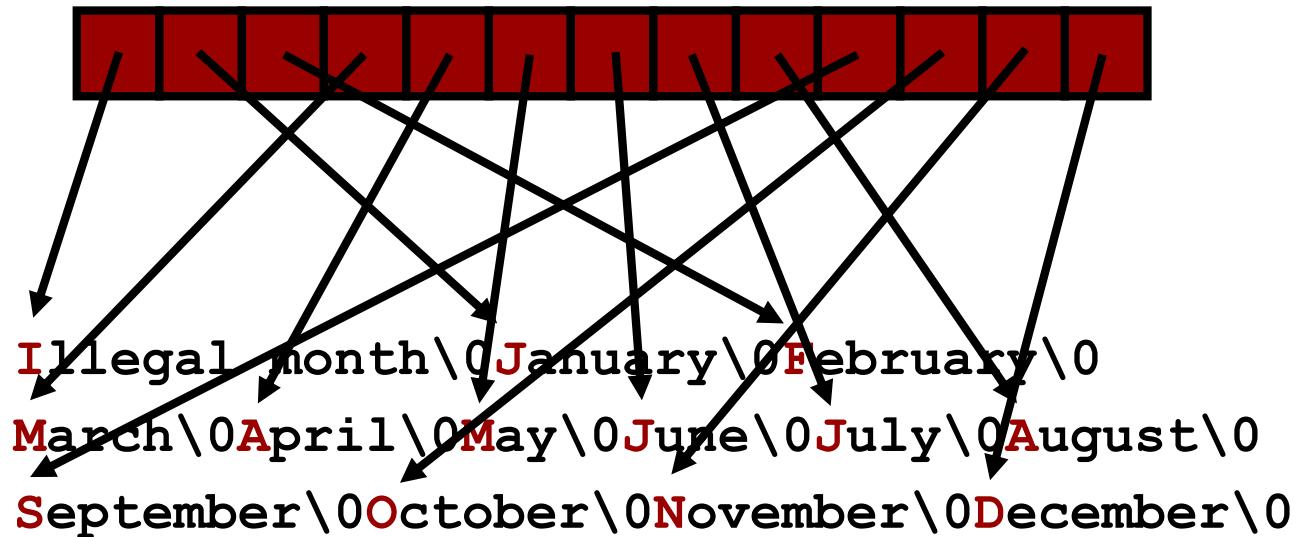
- An array is created in the static data area which will hold 13 character pointers, since there were 13 string constants



```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

# Arrays of Pointers

- ☞ The pointers are initialized like so



# Arrays of Pointers

So when the function is called:

```
printf("The third month is %s\n", month_name(3));
```

The line

```
return (n<1 || n>12) ? name[0] : name[n];
```

Gives us back the address (a pointer to) the 'M'



```
Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0
```

A marginally indexed array takes

- A. more space than a two-dimensional array
- B. less space than a two-dimensional array
- C. the same amount of space as a two-dimensional array
- D. It depends on whether all the items in the second dimension of the array are the same or different sizes



Today's number is 90,001

# Multi-Dimensional Arrays



# Definition

```
int ia[3][4];
```

Type

Address

Number  
of Columns

Number  
of Rows

Defined at compile time  
i.e. size must be known

# How does a two dimensional array work?

|   |   |   |   |   |
|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 |
| 0 |   |   |   |   |
| 1 |   |   |   |   |
| 2 |   |   |   |   |

How would you store it?

|   |   |   |   |   |
|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 |
| 0 |   |   |   |   |
| 1 |   |   |   |   |
| 2 |   |   |   |   |

## Column Major Order

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0,0 | 1,0 | 2,0 | 0,1 | 1,1 | 2,1 | 0,2 | 1,2 | 2,2 | 0,3 | 1,3 | 2,3 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Column 0

Column 1

Column 2

Column 3

## Row Major Order

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

Row 0

Row 1

Row 2

How would you store it?

# Advantage

- Using Row Major Order allows easy visualization as an array of arrays

**ia[1]**



**ia[1][2]**



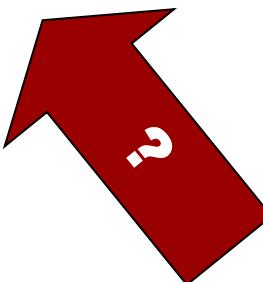
# What's the Output?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 int a[5][7];
 printf("sizeof a = %d\n", sizeof a);
 printf("sizeof a[3] = %d\n", sizeof a[3]);
 printf("sizeof a[3][4] = %d\n", sizeof a[3][4]);
 return EXIT_SUCCESS;
}
```

- 1) 28
- 2) 20
- 3) error
- 4) 4
- 5) 12

Recall: The type of a[3][4] is **int**



# What's the Output?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 int a[5][7];
 printf("sizeof a = %d\n", sizeof a);
 printf("sizeof a[3] = %d\n", sizeof a[3]);
 printf("sizeof a[3][4] = %d\n", sizeof a[3][4]);
 return EXIT_SUCCESS;
}
```

- 1) 28
- 2) 20
- 3) error
- 4) 5
- 5) 7

?

Recall: The type of a[3] is **array[7] of int**

# What's the Output?

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
 int a[5][7];
 printf("sizeof a = %d\n", sizeof a);
 printf("sizeof a[3] = %d\n", sizeof a[3]);
 printf("sizeof a[3][4] = %d\n", sizeof a[3][4]);
 return EXIT_SUCCESS;
}
```

- 1) 28
- 2) 35
- 3) error
- 4) 140
- 5) 280



?

Recall: The type of a is **array[5] of array[7] of int**

- ↗ One Dimensional Array

```
int ia[6];
```

- ↗ Address of beginning of array:

```
ia ≡ &ia[0]
```

- ↗ Two Dimensional Array

```
int ia[3][6];
```

- ↗ Address of beginning of array:

```
ia ≡ &ia[0][0]
```

- ↗ also

- ↗ Address of row 0:

```
ia[0] ≡ &ia[0][0]
```

- ↗ Address of row 1:

```
ia[1] ≡ &ia[1][0]
```

- ↗ Address of row 2:

```
ia[2] ≡ &ia[2][0]
```

# Element Access

- Given a row and a column index
- How to calculate location?
- To skip over required number of rows:

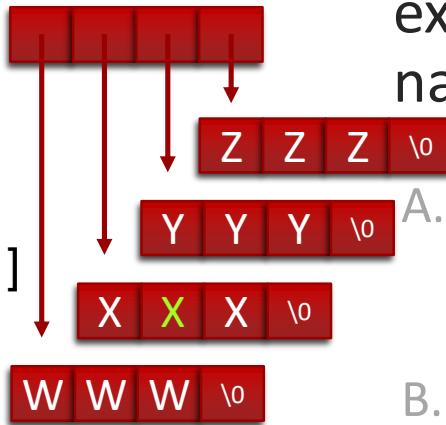
**`row_index * sizeof(row)`**

- This plus the *address of array* gives the address of first element of desired row
- Add **`column_index * sizeof(arr_type)`** to get actual desired element

|     |     |     |     |     |     |     |     |     |     |     |     |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0,0 | 0,1 | 0,2 | 0,3 | 1,0 | 1,1 | 1,2 | 1,3 | 2,0 | 2,1 | 2,2 | 2,3 |
|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|



char \*a[];



Given the **declarations** “char \*a[]” and “char b[][5]”, why do the expressions “a[1][1]” and “b[1][1]” appear to do the same thing, but navigate completely different data structures?

char b[][5];



- A. Because “a[1]” is of type “pointer to char”, it yields a pointer which happens to point to the first character of a string
- B. Because “b[1]” is of type “array[5] of char”, in an expression it is promoted to a pointer to the first element of a 5 char array.
- C. Because sizeof(a[1]) is 8 (assuming 64-bit pointers) and sizeof(b[1]) is 5.
- D. All of the above. ←

# Pointer Calculations on Our Own

- ↗ So you want to compute array memory addresses yourself?
- ↗ Cast the pointer to `(char *)`
- ↗ The multiplier applied to arithmetic on that pointer will then be `sizeof(char) == 1`
- ↗ So to access element  $(r, c)$  in *arr*:

```
int arr[5][10];
int offset = (r * 10 + c) * sizeof(int);
int *p = (int *)((char *)arr + offset);
```
- ↗ *p* points to the address of *arr[r][c]*
- ↗ Where would *p+1* point?
- ↗ What happened to type checking??

# Multidimensional Array Parameters

```
void tester(int arr[][][4][5], int len)
{ ... }
int main()
{
 int ia[3][4][5];
 int ib[8][4][5];
 int *ic;
// ic = ia; // warning: incompatible type
 ic = &ia[0][0][0];

 tester(ia, 3);
 tester(ib, 8);
 tester((int (*)[4][5])ic, 3);
}
```

# What's Up?

- ↗ What's up with `int arr[][][4][5]???`
- ↗ Consider a one-dimensional array declaration, e.g. "double q[];"
- ↗ **Declarations** of 1D arrays don't need to be told the size. Why?
- ↗ If asked to calculate the memory address of a given element, does one need to know the full size of the array?
  
- ↗ Consider a 2D array
- ↗ What is needed to calculate the address of a given element (i,j)?  
  
$$\text{offset} = i * \text{columns} + j$$
- ↗ Note we don't need **rows** for this calculation!
- ↗ We never need the first dimension to **declare** an array

The first dimension of an multi-dimensional array type **declaration** can be optionally omitted

- A. It can't be omitted
- B. only when it is a formal parameter to which a pointer is passed
- C. because a declaration does not allocate storage and because C does not check array bounds, the compiler doesn't need to know how long the array is to calculate the position of its elements
- D. because C array bounds checking only works when the array is local to a program scope



Now think about

- ↗ A 3D array



**int a**

Now think about

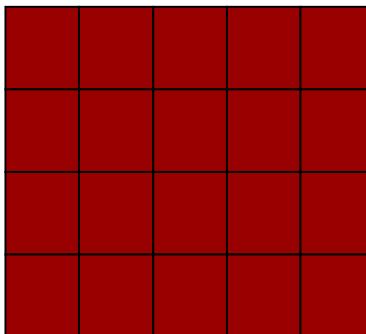
- ↗ A 3D array



**int a[5]**

Now think about

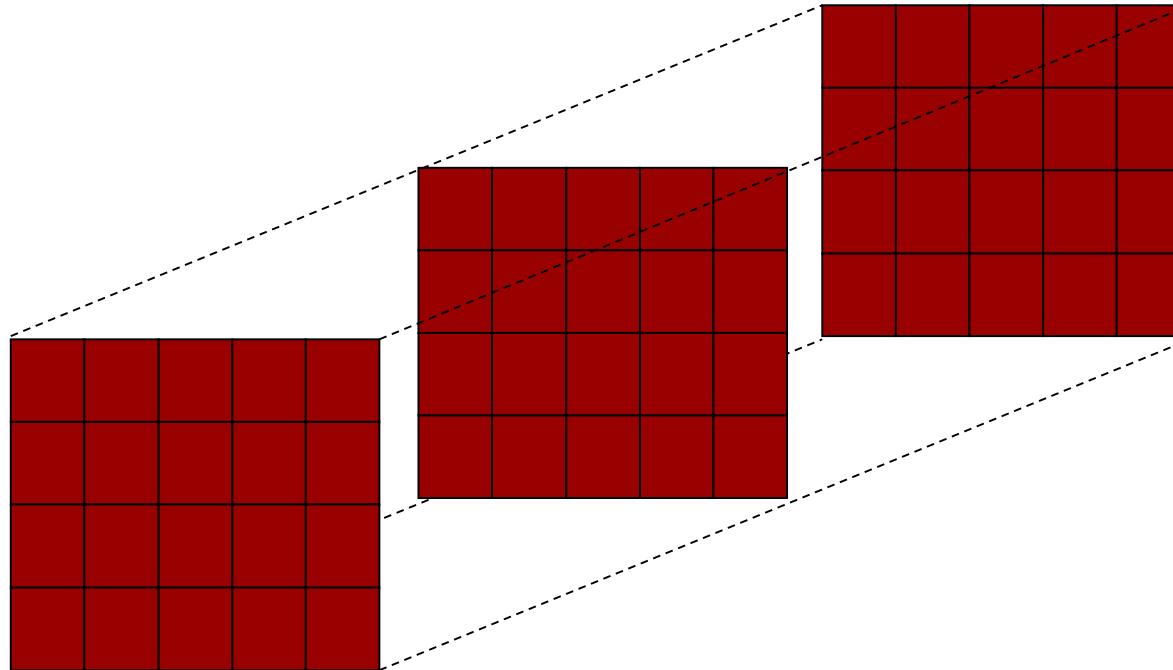
- ↗ A 3D array



`int a[4][5]`

Now think about

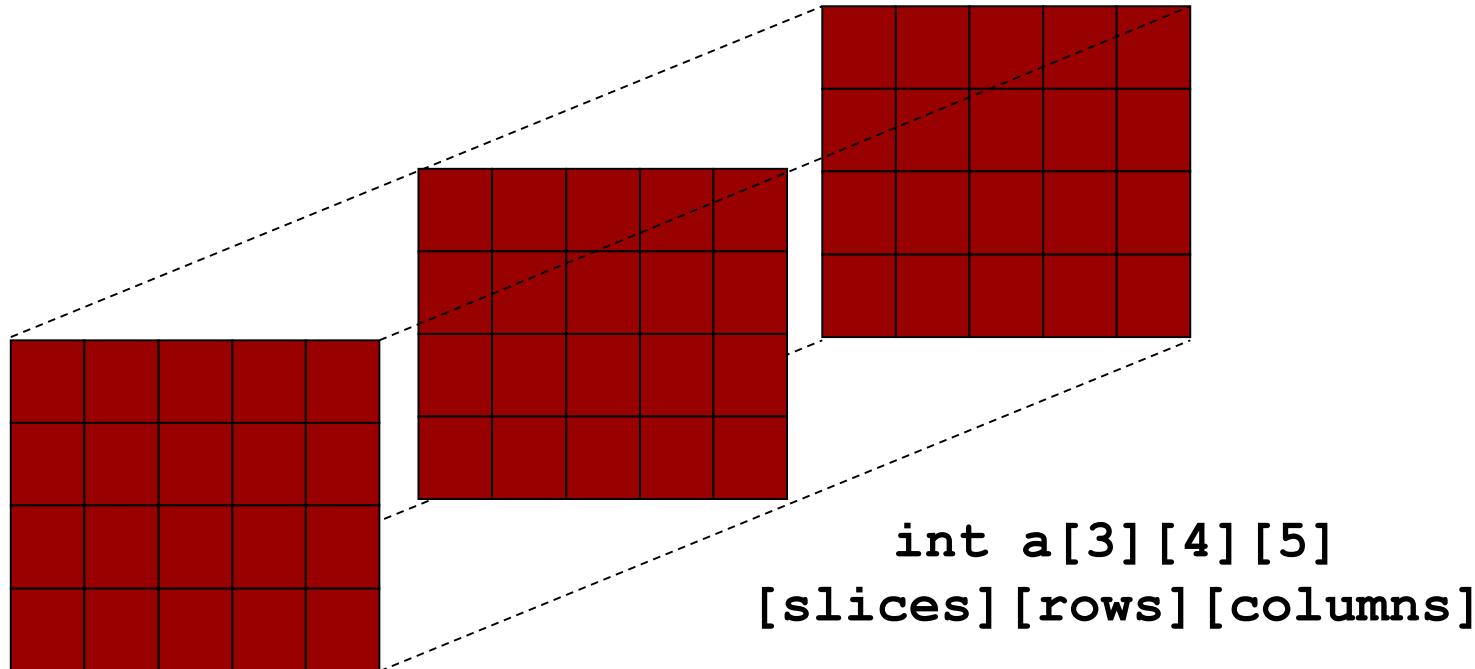
↗ A 3D array



`int a[3][4][5]`

# Offset to $a[i][j][k]$ ?

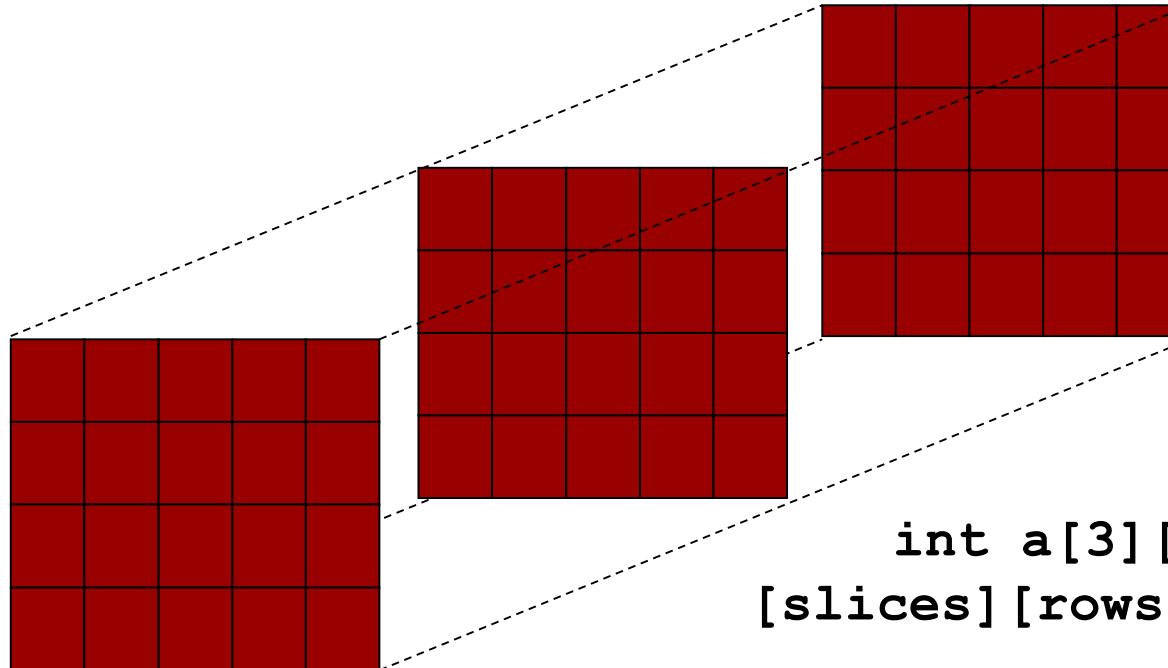
↗ A 3D array



**offset = ( $i * \text{rows} * \text{columns}$ ) + ( $j * \text{columns}$ ) +  $k$**   
*(Do you see “slices” used anywhere?)*

# Offset to $a[i][j][k]$ ?

↗ A 3D array



```
int a[3][4][5]
[slices] [rows] [columns]
```

**offset = ( $i * \text{rows} * \text{columns}$ ) + ( $j * \text{columns}$ ) +  $k$**   
 $a[i][j][k] == *(*(*(\text{a} + i) + j) + k)$

# More Detailed C Topics



# More on Structures



- ↗ Recall

- ↗ Collection of items which may be of different types

- ↗ Analogous to

- ↗ Records in Pascal
  - ↗ Classes (with just variables) in Java

- ↗ In C, structs

- ↗ Use named, not structural, type equivalence
  - ↗ Untagged structs are each a different type

# Struct Declaration/Definition

```
struct [<optional tag>] [{
 <type declaration>;
 <type declaration>;
 ...
}] [<optional variable list>];
```

- Struct declarations that have a **member list** in curly braces define a **new type**, specifically a struct type.
- If the optional tag is omitted it creates an unnamed struct type that's different from every other struct type.

# Struct Name Scopes

```
struct [<optional tag>] [{
 <type declaration>;
 <type declaration>;
 ...
}] [<optional variable list>];
```

- Filling in <optional variable list> often makes the declaration also a definition; the variables defined appear in the function's name scope just like any other variable.
- Struct tags are in a **separately-scoped name space** from variables, i.e. a struct variable can have the same name as a struct tag without causing confusion
- Struct member names are in yet another name space local to the structure type, e.g. every structure type could have a member named “next”

# Initialization

```
struct mystruct_tag {
 int myint;
 char mychar;
 char mystr[20];
};

struct mystruct_tag ms = {42, 'f', "goofy"};
```

# Question

```
struct mystruct_tag {
 int myint;
 char mychar;
 char mystr[20];
} ms;
ms.mystr = "foo";
```

- A. The assignment is legal
- B. The assignment is illegal because “foo” is a different size than ms.mystr
- C. The assignment is illegal because “foo” is stored in the constant data section of memory
- D. The assignment is illegal because it is trying to assign an array to an array



# Question

```
struct mystruct_tag {
```

```
 int myint;
```

```
 char mychar;
```

```
 char mystr[20];
```

```
};
```

```
struct mystruct_tag ms = {42, 'b', "Boo!"};
```

- A. The initializer is legal because a character array can be initialized to a string as a special case 
- B. The initializer is illegal because “Boo!” is a different size than ms.mystr
- C. The initializer is illegal because “Boo!” is stored in the constant data section of memory
- D. The initializer is illegal because it is trying to assign an array to an array

# Copying Structs

```
struct s {
 int i;
 char c;
} s1, s2;

s1.i = 42;
s1.c = 'a';
s2 = s1;
s1.c = 'b' ;
s2.i contains ?
42
s2.c contains ?
a ←
```

Note that assigning the structure just copied the bytes from one memory block to another. There is no connection between them.

# Copying Structs

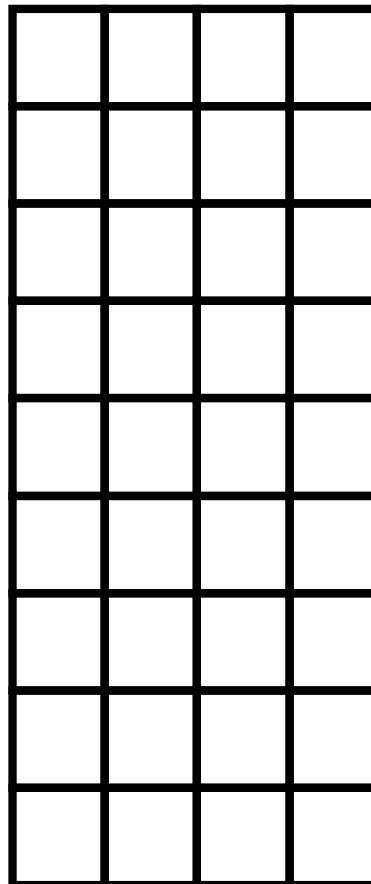
```
struct s {
 int i;
 char c[8];
} s1, s2;

s1.i = 42;
strcpy(s1.c, "foobar");
s2 = s1; ←
s2.i contains 42
s2.c contains:
'f', 'o', 'o', 'b', 'a', 'r', '\0', ??
```

Since we assigned s1 to s2,  
s2.c is going to contain  
exactly the same characters  
as are in s1.c, even  
including the unspecified  
character at s1.c[7]!

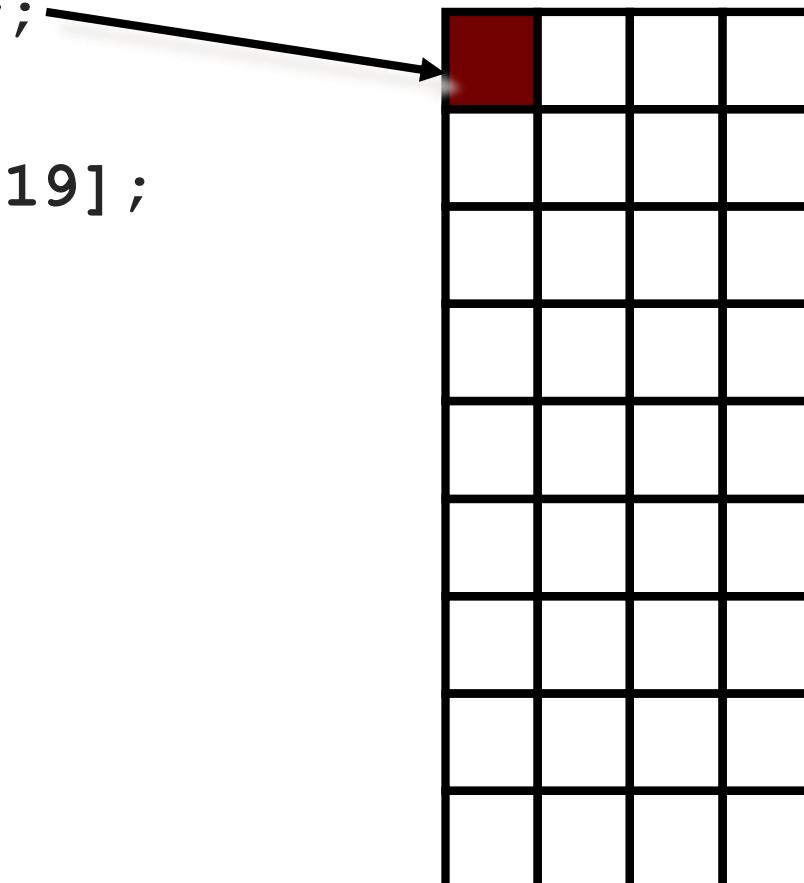
# Where Do Members Get Stored in Memory?

```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;
```



# Where Do Members Get Stored in Memory?

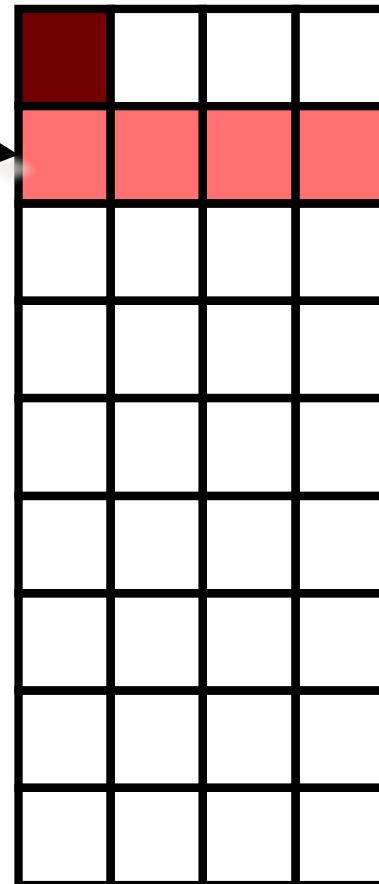
```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;
```



# Where Do Members Get Stored in Memory?

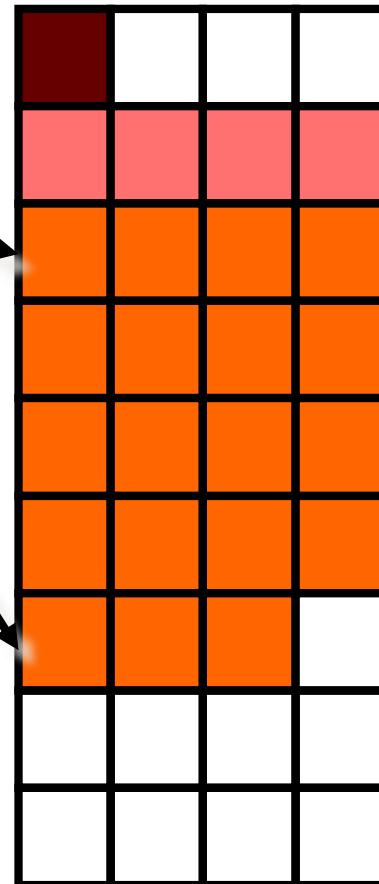
```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;
```

C can't reorder members,  
but it can add filler to  
preserve alignment



# Alignment Rules are Respected

```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;
```

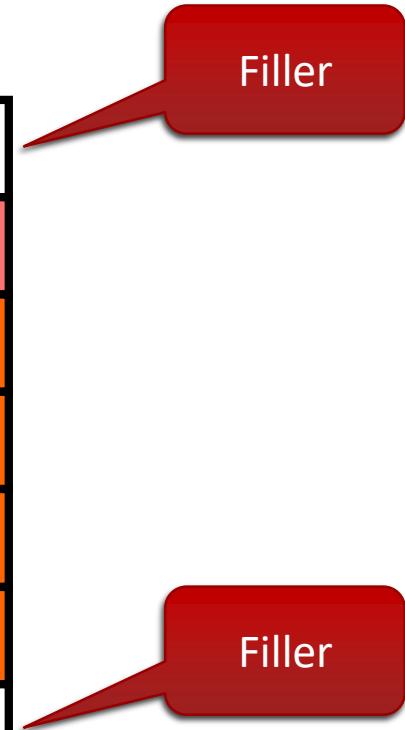
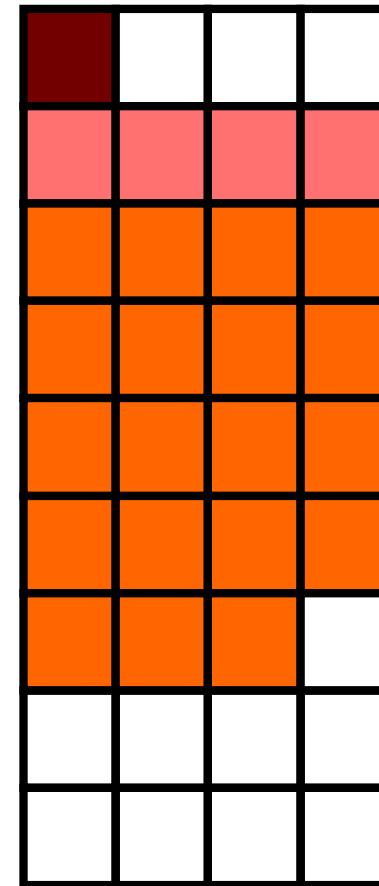


# What is sizeof(mystruct)?

```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;
```

Structs are usually filled out to  
meet the most stringent  
alignment of its members

$\text{sizeof(mystruct)} = 28$



# We Still Use base+offset !

- The compiler keeps track of the offsets of each member in a table for each struct

| <u>member</u> | <u>offset</u>                                         |
|---------------|-------------------------------------------------------|
| mychar        | 0                                                     |
| myint         | 4                                                     |
| mystr         | 8 (Sum of sizes of all previous elements incl filler) |

Question: Assume "mystruct" is located at location 1000  
What will be address of mychar, myint and mystr?

1000, 1004, 1008

# Don't Believe Me? Ask the Compiler!

```
#include <stdio.h>

struct {
 char mychar;
 int myint;
 char mystr[19];
} mystruct;

int main() {
 printf("Address of mystruct = %p\n", (void *)&mystruct);
 printf("Offset of mychar = %ld\n",
 (void *)&mystruct.mychar - (void *)&mystruct);
 printf("Offset of myint = %ld\n",
 (void *)&mystruct.myint - (void *)&mystruct);
 printf("Offset of mystr = %ld\n",
 (void *)mystruct.mystr - (void *)&mystruct);
 printf("Size of mystruct = %ld\n", sizeof(mystruct));
}
```

# Don't Believe Me? Ask the Compiler!

```
$ gcc sizes.c
$./a.out
Address of mystruct = 0x10ae74018
Offset of mychar = 0
Offset of myint = 4
Offset of mystr = 8
Size of mystruct = 28
```

- Can we say:

```
struct {
 char mychar;
 int myint;
 char mystr[19];
} mystuct;
mystuct.mystr[4] = 'x' ;
```

- Yes we can!

# Base+offset again!

```
mystruct.mystr[4] = 'x' ;
```

- ↗ How do we get to the right character?
  - ↗ First find the address of the struct member
    - ↗ &mystruct + offset to mystr → &mystruct + 8
  - ↗ Then find the location within the struct member (if needed)
  - ↗ Using the type of the member, find the offset to the desired element
    - ↗ Offset of char mystr[4] → 4 \* sizeof(char)
  - ↗ Add them: &mystruct + 8 + 4
- ↗ Example
  - ↗ mystruct is located at 2000 1. 2005
  - ↗ Address of mystr is 2000 + 8 2. 2010
  - ↗ So element 4 of mystr is offset by 4 \* sizeof(char) 3. 2012
  - ↗ The address of mystruct.mystr[4] is... 4. 2014
  - ↗ ...2000 + 8 + 4 =

# The Calculation in Detail

|                                           |                |
|-------------------------------------------|----------------|
| Base address of mystruct                  | &mystruct      |
| Offset of mystr within structure          | 8              |
| Offset of element 4 within mystruct.mystr | 4              |
|                                           | &mystruct + 12 |

```
mystruct.mystr[4] = 'x';
```

- If mystruct is at address 2000, then mystruct.mystr[4] will be at address 2012.

# Arrays of structs?

```
struct m astruct[25];

astruct[6].mystr[3] = 'y';
```

- ↗ How do we get to the right character?
  - ↗ First find the address of the struct member
    - ↗ &astruct + offset to astruct[6] + offset to mystr → &astruct + 6\*sizeof(struct m) + 8
  - ↗ Then find the location within the struct member (if needed)
  - ↗ Using the type of the member, find the offset to the desired element
    - ↗ Offset of char mystr[3] is 3 \* sizeof(char)
  - ↗ Add them: &astruct + 6\*28 + 8 + 3
- ↗ Example
  - ↗ astruct is located at 2000
  - ↗ Address of mystr is 2000 + 6\*28 + 8
  - ↗ So element 3 of mystr is offset by 3 \* sizeof(char)
  - ↗ The address of astruct.mystr[3] is...
    - ↗ ...2000 + 6\*28 + 8+ 3 =

1. 2176

2. 2177

3. 2179

4. 2181

# Detailed Calculation

|                                           |                           |
|-------------------------------------------|---------------------------|
| Base address of mystruct                  | mystruct                  |
| Offset of element 6 of mystruct           | $6 * 28$                  |
| Offset of mystr within structure          | 8                         |
| Offset of element 3 within mystruct.mystr | 3                         |
|                                           | $\&\text{mystruct} + 179$ |

```
struct foo mystruct[25];
```

```
mystruct[6].mystr[3] = 'y';
```

- ↗ If mystruct is at memory location 2000, then mystruct[6].mystr[3] is at location 2179

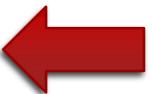
# Question

```
static struct {
 int n;
 char m[3];
 double p;
} s[12];
```

If *s* is stored at memory address 0x0e3c,  
where is *s[5].m[2]* stored?

- A. 0xe91
- B. 0xe92
- C. 0xe96
- D. 0xea0

- Member offsets  
*n*: 0, *m*: 4, *p*: 8
- Struct size  
16
- Offset from *s* to *&s[5]*  
 $5 * \text{sizeof}(s[0]) = 80 = 0x50$
- Offset from *&s[5]* to *s[5].m*  
 $4 = 0x4$
- Offset to from *s[5].m* to *&s[5].m[2]*  
 $2 * \text{sizeof}(\text{char}) = 2 = 0x2$
- Address  
 $0x0e3c + 0x50 + 0x4 + 0x2 = 0xe92$



## Structures may

- ↗ be copied or assigned
- ↗ have their address taken with &
- ↗ have their members accessed
- ↗ be passed as arguments to functions
- ↗ be returned from functions

## Structures may not

- ↗ be compared

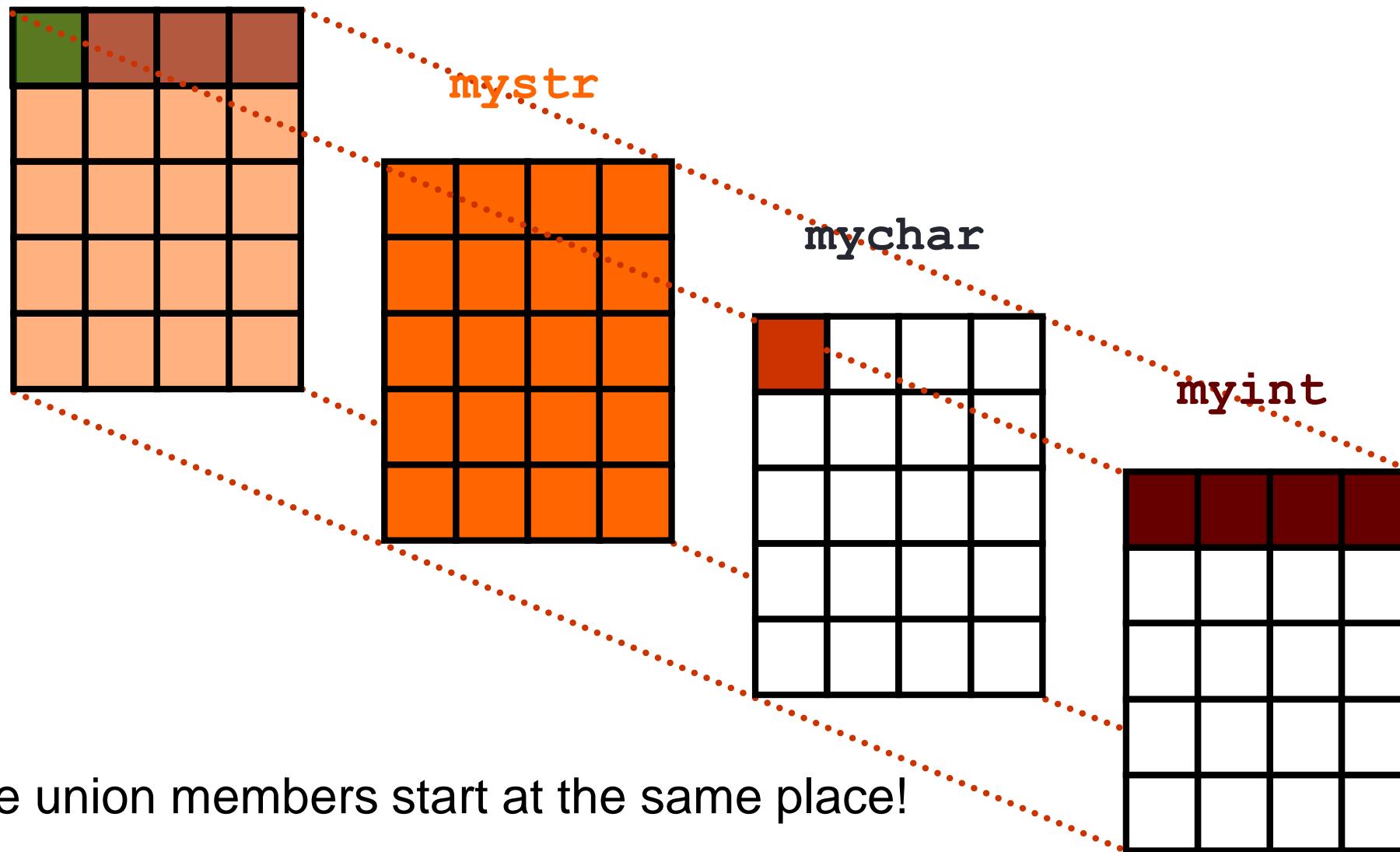
# Continuing with C: Part 2

- Unions
- Function Pointers

```
union {
 int myint;
 char mychar;
 char mystr[20];
} myun;
```

- ↗ Looks like a struct
- ↗ and the access is the same as a struct
- ↗ So what's the difference?
- ↗ All of the members have an offset of zero – that's it!

# Unions



```
union {
 int myint;
 char mychar;
 char mystr[20];
} myun;
```

- ↗ &myun.myint == &myun.mychar == &myun.mystr[0]
- ↗ And sizeof(myun) is the size of the largest member
  
- ↗ Effectively all items in a union "start" at the same place
- ↗ But why?

# Unions and base+offset

- Compiler keeps track of offsets into structure of each member in "some sort" of symbol table

| <u>member</u> | <u>offset</u> |
|---------------|---------------|
| myint         | 0             |
| mychar        | 0             |
| mystr         | 0             |

Question: Assume "mystruct" is located at location 1000  
What will be address of myint, mychar and mystr?

1000, 1000, 1000

- ↗ Suppose we want to store information about athletes
- ↗ For all we want
  - ↗ Name, JerseyNum, Team, Sport
- ↗ For football players we want
  - ↗ Attempts, yards, TDs, Interceptions, etc.
- ↗ For baseball players we want
  - ↗ Wins, Losses, Innings, ERA, Strikeouts, etc.
- ↗ For basketball players we want
  - ↗ Shots, Assists, Rebounds, Points, etc.

We code:

```
struct player {
 char name[20];
 char jerseynum[4];
 char team[20];
 int player_type;
 union sport {
 struct football {...} footbstats;
 struct baseball {...} basebstats;
 struct basketball {...} baskbstats;
 } thesport;
} theplayer;

theplayer.sport.footbstats.tds = 3;
```

- Often used in implementing the polymorphism found in object-oriented languages

Unions may

## Unions may

- ↗ be copied or assigned
- ↗ have their address taken with &
- ↗ have their members accessed
- ↗ be passed as arguments to functions
- ↗ be returned from functions
- ↗ be initialized (but only the first member)

## Unions may not

- ↗ be compared

# Function Pointers



# If we want a function pointer

- ↗ To store the address of a variable we use a pointer (e.g. `int *ip;`)
- ↗ Same is true for holding the address of a function:

```
int fi(void); /* Function that returns an int */
int *fpi(void); /* Function that returns a pointer
 * to an int */
int (*pfi)(void); /* pfi is a pointer to a function
 * returning int! */
```

# Recall

## Using it...

```
int fi(void); /* Function that returns an int */
int *fpi(void); /* Function that returns a
 /* pointer to an int */
int (*pfi)(void); /* Declaring pfi to be a
 * pointer to a function!
pfi = fi; /* Legal assignment
pfi = fi(); /* NO */
```

➤ Notice similarity to

```
int ia[10];
int *ip;
ip = ia;
```

# But what good is a function pointer?

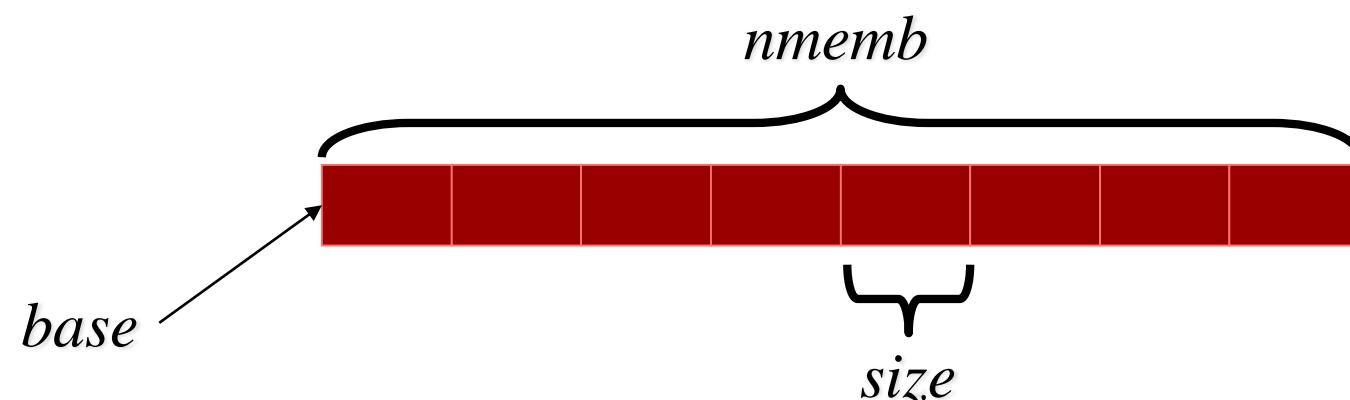
- ↗ Say you are writing a general purpose sorting function.
- ↗ You want it to be able to sort anything
  - ↗ Numbers
  - ↗ Strings
  - ↗ Structs, Unions, and other stuff
- ↗ Obviously comparing numbers, strings, and other stuff calls for at least two different techniques
- ↗ What if we write functions that do the comparison we need
  - ↗ A function to compare numbers
  - ↗ A function to compare strings
  - ↗ A function to compare other stuff

# But what good is a function pointer?

- Now when we call the function to do the sorting we pass in a pointer to the appropriate function for the type of data we have!
- "But wait," I hear you say, "It would be easier to write my own sorting function!"

**qsort(3)****qsort(3)****NAME****qsort - sort an array****ANSI\_SYNOPSIS**

```
#include <stdlib.h>
void qsort(void * base, size_t nmemb, size_t size,
 int (* compar)(const void *, const void *));
```



**DESCRIPTION**

**qsort** sorts an array (beginning at **base**) of **nmemb** objects.  
**size** describes the size of each element of the array.

You must supply a pointer to a comparison function, using the argument shown as **compar**. (This permits sorting objects of unknown properties.) Define the comparison function to accept two arguments, each a pointer to an element of the array starting at **base**. The result of **(\*[compar])>>** must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where ``less than'' and ``greater than'' refer to whatever arbitrary ordering is appropriate).

The array is sorted in place; that is, when **qsort** returns, the array elements beginning at **base** have been reordered.

**qsort(3)**

**qsort(3)**

**RETURNS**

**qsort does not return a result.**

**PORTABILITY**

**qsort is required by ANSI (without specifying the sorting algorithm).**

**SOURCE**

**[src/newlib/libc/stdlib/qsort.c](#)**

# QSort Demo

```
#include <stdlib.h>

void qsort (
 void * base,
 size_t nmemb,
 size_t size,
 int (* compar) (const void *, const void *)
);
```

The result of "compar" must be negative if the first argument is less than the second, zero if the two arguments match, and positive if the first argument is greater than the second (where "less than" and "greater than" refer to whatever arbitrary ordering is appropriate).

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100
int compar_ints(const void *pa, const void *pb) {
 return *((int *)pa) - *((int *)pb);
}

int compar_strings(const void *ppa, const void *ppb) {
 return strcmp(*((char **)ppa) , *((char **)ppb));
}
```

```
define MAX 100
int main(int argc, char **argv) {
 char *strings[] = {"dec", "sun", "ibm", "apple", "hp",
"ti", "univac"};
 int i, s;
 int a[MAX];
 if(argc == 2 && *(argv[1]) == 'a') {
 s = sizeof(strings)/sizeof(strings[0]);
 qsort(strings, s, sizeof(strings[0]),
 compar_strings);
 for(i = 0; i < s; i++) {
 printf(" %s", strings[i]);
 }
 printf("\n");
 }
else...
```

```
 else {
 for(i = 0; i < MAX; i++) {
 a[i] = rand() % 100;
 printf(" %d", a[i]);
 }
 printf("\n\n");
 qsort(a, MAX, sizeof(int),
 compar_ints);
 for(i = 0; i < MAX; i++)
 printf(" %d", a[i]);
 }
 return 0;
}
```

# 7<sup>th</sup> Ed Unix Device Driver Table

Major, minor device numbers

```
$ ls -l /dev
```

```
total 0
```

|            |   |      |          |     |    |     |    |       |         |
|------------|---|------|----------|-----|----|-----|----|-------|---------|
| crw-----   | 1 | dan  | staff    | 0,  | 0  | Mar | 13 | 05:15 | console |
| crw-rw-rw- | 1 | root | wheel    | 3,  | 2  | Mar | 13 | 13:59 | null    |
| crw-----   | 1 | root | wheel    | 11, | 0  | Feb | 9  | 15:00 | pf      |
| crw-rw-rw- | 1 | root | tty      | 15, | 16 | Mar | 13 | 13:59 | ptmx    |
| crw-rw-rw- | 1 | root | wheel    | 5,  | 0  | Feb | 9  | 15:00 | ptyp0   |
| crw-rw-rw- | 1 | root | wheel    | 5,  | 1  | Feb | 9  | 15:00 | ptyp1   |
| crw-rw-rw- | 1 | root | wheel    | 14, | 0  | Feb | 9  | 15:02 | random  |
| crw-r----  | 1 | root | operator | 1,  | 0  | Feb | 9  | 15:00 | rdisk0  |
| crw-r----  | 1 | root | operator | 1,  | 4  | Feb | 9  | 15:00 | rdisk1  |
| crw-r----  | 1 | root | operator | 1,  | 7  | Feb | 9  | 15:00 | rdisk2  |
| crw-rw-rw- | 1 | root | wheel    | 2,  | 0  | Feb | 23 | 10:35 | tty     |
| crw-rw-rw- | 1 | root | wheel    | 4,  | 0  | Feb | 9  | 15:00 | ttyp0   |
| crw-rw-rw- | 1 | root | wheel    | 4,  | 1  | Feb | 9  | 15:00 | ttyp1   |
| crw-rw-rw- | 1 | root | wheel    | 4,  | 2  | Feb | 9  | 15:00 | ttyp2   |
| crw-rw-rw- | 1 | root | wheel    | 3,  | 3  | Feb | 9  | 15:00 | zero    |

# Defining the Table Struct

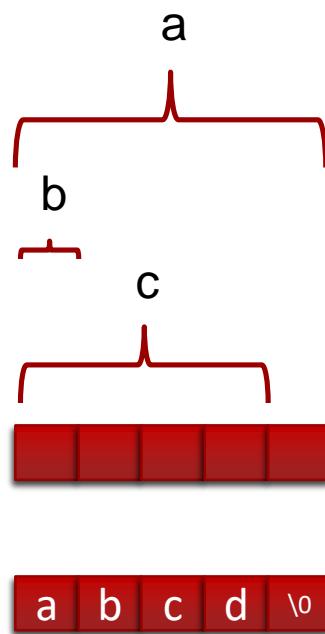
```
extern struct cdevsw
{
 int (*d_open)();
 int (*d_close)();
 int (*d_read)();
 int (*d_write)();
 int (*d_ioctl)();
 int (*d_stop)();
 struct tty *d_ttys;
} cdevsw[];
```

Major device number is the index to this table  
Minor device number is passed on to the function

```
struct cdevsw cdevsw[] =
{
 klopen, klclose, klread, klwrite, klioctl, nulldev, 0, /* console = 0 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* pc = 1 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* lp = 2 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dc = 3 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dh = 4 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dp = 5 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dj = 6 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* dn = 7 */
 nulldev, nulldev, mmread, mmwrite, nodev, nulldev, 0, /* mem = 8 */
 nulldev, nulldev, rkread, rkwrite, nodev, nulldev, 0, /* rk = 9 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rf = 10 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rp = 11 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* tm = 12 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* hs = 13 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* hp = 14 */
 htopen, htclose, htread, htwrite, nodev, nulldev, 0, /* ht = 15 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* du = 16 */
 syopen, nulldev, syread, sywrite, sysioctl, nulldev, 0, /* tty = 17 */
 nodev, nodev, nodev, nodev, nodev, nulldev, 0, /* rl = 18 */
 0
};
```

Questions?

# Question



Given execution of the following code, what is the value of m.a?

```
union m {
 char a[5];
 char b;
 int c;
} m;
strcpy(m.a, "abcd");
```

**m.c = 0;**

**m.b = 'q';**

- A. m.a contains "q" ←
- B. m.a contains "abcd"
- C. m.a contains an empty string
- D. m.a contents are unknown

# Question

What kind of value will you typically find in a pointer to a function?

- A. The address of the function arguments on the stack
- B. The address of the first word of the function code
- C. An integer describing the success or failure of the function
- D. A string containing the function's name



# Dynamic Allocation

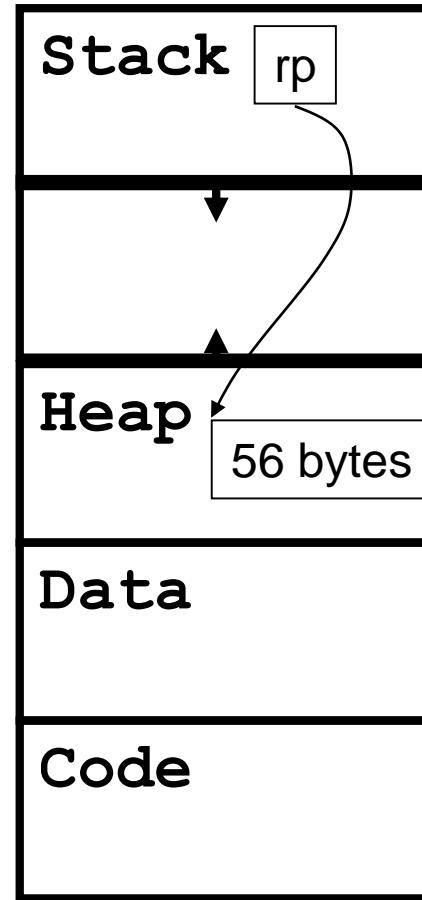


# Dynamic Storage

- Example: We need space to allow the user to enter some data, but how much?
  - 80 bytes? 132? 1? 256? 10000?
- Two Friends from the C library: #include <stdlib.h>
  - malloc()
  - free()
- These friends create and manage the heap for us, allowing us to grab storage at run time

# malloc()

```
struct r {
 char name[40];
 double gpa;
 struct r *next;
};
struct r *rp;
rp = malloc(sizeof(struct r));
if (rp == NULL) {
 /* Handle Error! */
}
↗ Options for handling error
↗ Abort
↗ Ask again
↗ Save user data
↗ Ask for less
↗ Free up something
```



# Don't Do This!

```
rp = malloc(sizeof struct r);
// Code foolishly inserted here!
if(rp == NULL)
```

- ↗ Be afraid.
- ↗ Using **rp** before you check it is a crash waiting to happen.

# Idiomatic But Safe

```
if((rp = malloc(sizeof struct r)) == NULL) {
 /* Handle Error Here */
}
```

- ↗ This is the way it's done in real life.
- ↗ Don't mix up the = and == operators!
  
- ↗ Alternate syntax (*because NULL==0*):

```
if(!(rp = malloc(sizeof struct r))) {
 /* Handle Error Here */
}
```

# Escape From Type-Checking

- ↗ malloc() returns a pointer to at least as many bytes as we requested.
- ↗ But what is the type of this pointer and how do we use it to store items of a different type?
- ↗ malloc() is declared as “void \*malloc(unsigned long)”
- ↗ C uses the idiom “pointer to void” for a generic pointer
- ↗ To be safe, you should cast this pointer into the correct type so that type-checking can work for you again!

```
int *ip;
ip = (int *)malloc(sizeof int);
```

- ↗ Otherwise, the compiler will silently cast your “void \*” pointer into any other kind of pointer without checking

# Done With a Chunk of Storage

- When you're done with a chunk of storage, you use **free()** to make it available for reuse.
- Remember, C doesn't do garbage collection
- From our previous example

```
free(rp);
```

returns the memory back to the heap for re-use by someone else

- You **must not** use the value in rp after the call to **free()**, nor may you dereference the memory it points to!
- There's no guarantee what's at \*rp after you call **free()** – assume it is garbage data!
- With modern C libraries, **free(NULL)** does nothing but isn't an error

- From our previous example

```
free(rp);
```

- The variable rp still exists.
  - It is a pointer to struct r
- But what it points to is now garbage data.
  - We should never dereference rp again: \*rp
- We can, however, assign a new value to rp
  - That is okay – make it point somewhere else
- The compiler will NOT help you with this.
  - Mistakes will cause run-time errors

# Other memory allocation functions

- `void *malloc(size_t n);`
  - Allocates (at least) n bytes of memory on the heap, returns a pointer to it
  - Assume memory contains garbage values
- `void *calloc(size_t num, size_t size);`
  - Allocates (at least) num\*size bytes of memory on the heap, returns a pointer to it
  - Memory will be zero'ed out.
- `void *realloc(void *ptr, size_t n);`
  - Relocates (at least) n bytes of memory on the heap, returns a pointer to it
  - Copies the data starting at ptr that was previously allocated
  - Often used to expand the memory size for an existing object on the heap

# Pointer Video

↗ <https://www.youtube.com/watch?v=5VnDaHBi8dM>

# Handling Persistent Data



```
char *foo(void)
{
 static char ca[10];
 return ca;
}
```

- Anyone calling this function now has access to ca in this block. Could be dangerous. Why?
  
- Note that this approach is not dynamic

# Example

```
char *strFromUnsigned(unsigned u)

{
 static char strDigits[] = "?????";

 char *pch;

 pch = &strDigits[5];

 do

 *--pch = (u % 10) + '0';

 while((u /= 10) > 0);

 return pch;
}
```

```
strHighScore =
 strFromUnsigned(HighScore);
```

.

.

.

```
strThisScore =
 strFromUnsigned(ThisScore);
```

```
char *foo(void)
{
 char ca[10];
 return ca;
}
```

- Since ca was allocated on stack during function call pointer returned is now pointing to who-knows-what

Bad

```
char *foo(void)
{
 char *ca = malloc(...);
 /* error checking but no free */
 return ca;
}
```

- This actually works, but the caller needs to know that they're responsible for the free()

# Memory Leaks

- Memory leaks occur when the programmer loses track of memory allocated by malloc or other functions that call malloc

```
void foo(void)
{
 char *ca = malloc(...);
 /* no free */
 return;
}
```

➤Bad

# Memory Management

- Some functions that call malloc
  - calloc
  - strdup
  - regcmp
  - others...
- C doesn't do automatic memory management for efficiency reasons
  - If you want to manage memory...do it yourself!

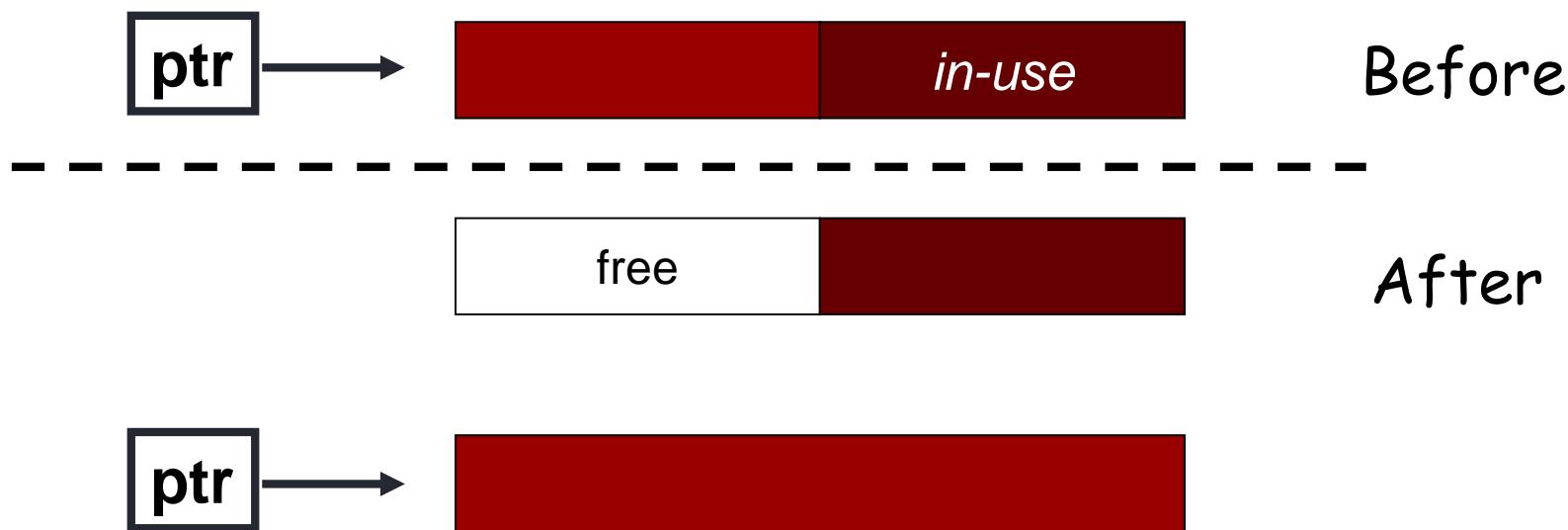
```
void *calloc(size_t num, size_t size);
```

- ↗ Call malloc() to find space for **num** new allocations
- ↗ Initialize the space to zero (0)
- ↗ Not much to discuss...

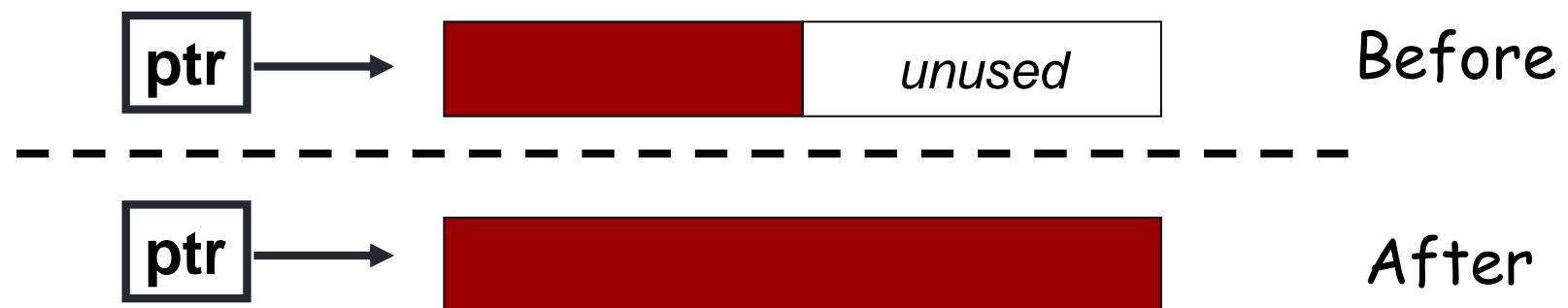
```
ptr = realloc(ptr, num_bytes);
```

- What it does (conceptually)
  - Find space for new allocation
  - Copy original data into new space
  - Free old space
  - Return pointer to new space

realloc()



# realloc: What might happen



- Realloc may return

- same pointer
- different pointer
- NULL

- Is this a good idea?

```
cp = realloc(cp, n);
```

1. Yes
2. No
3. Sometimes

- Is this a good idea?

```
cp = realloc(cp, n);
```

- No!
- If realloc returns NULL cp is lost
- Memory Leak!

# How to do it properly

```
void *tmp;

if((tmp = realloc(cp,...)) == NULL)

{

 /* realloc error */

}

else

{

 cp = tmp;

 free(tmp);

}
```



# Additional Edge Cases

- `realloc(NULL, n) ≡ malloc(n);`
- `realloc(cp, 0) ≡ free(cp); // only on some compilers`
- These can be used to make realloc work in a single loop design to build a dynamic structure such as a linked list.

# Example

```
int size = 0; /* Size of "array" */
int *ip = NULL; /* Pointer to "array" */
int *temp;
int i;
char buffer[80];
while(fgets(buffer, 80, stdin) != NULL) {
 size++;
 if((temp = realloc(ip, size*sizeof(*temp))) == NULL) {
 fprintf(stderr, "Realloc failure\n");
 exit(EXIT_FAILURE);
 }
 ip = temp;
 ip[size-1] = strtol(buffer, NULL, 10);
}
```

# Dynamic Allocation: What can go wrong?

- ↗ Allocate a block and lose it by losing the value of the pointer
- ↗ Allocate a block of memory and use the contents without initialization
- ↗ Read or write beyond the boundaries of the block
- ↗ Free a block but continue to use the contents
- ↗ Call realloc to expand a block of memory and then – once moved – keep using the old address
- ↗ FAIL TO NOTICE ERROR RETURNS

# Questions?



# Question

Is there anything wrong with the following code that frees all the nodes in a linked list?

```
for (struct node *p = Head; p != NULL; p = p->next)
 free(p);
```

- A. The head of the list doesn't get freed
- B. The code is satisfactory as it stands
- C. The reinitialization uses memory that has been freed
- D. The code uses free() multiple times on the same pointer

```
struct node *t;
for (struct node *p = Head; p != NULL; p = t) {
 t = p->next
 free(p);
}
```



# Malloc Implementation



## ➤ K&R Malloc Implementation

- Headers
- Heap Layout
- Allocating the Correct Amount of Memory
- **malloc ()** : Getting the memory for work
- **free ()** : Recycling the memory when done
- **morecore ()** : OS requests for real memory

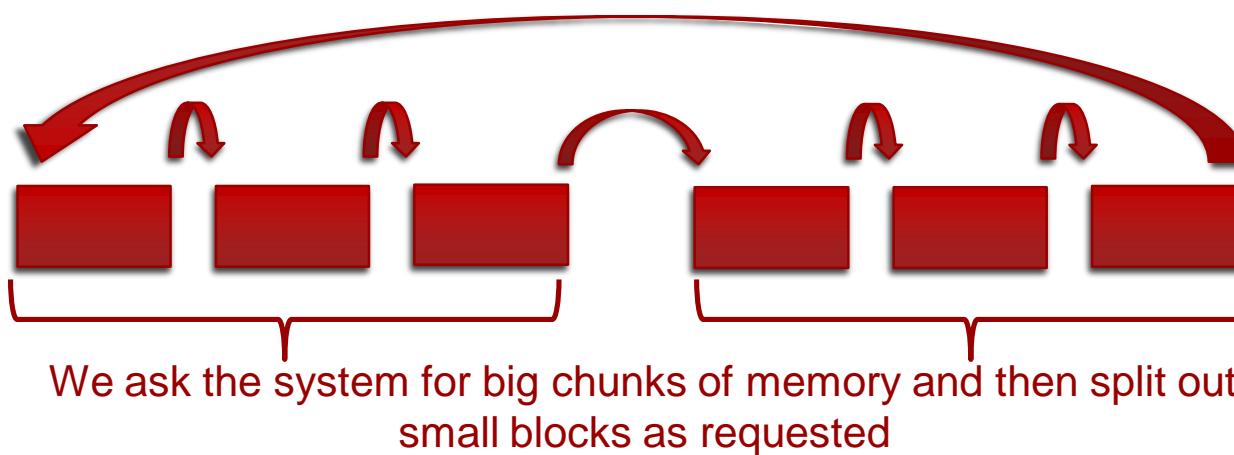
# Memory Management

- ↗ The K&R implementations of malloc() and free() are one of many ways to implement these.
- ↗ We've refactored the code a bit. It was written for compactness and efficiency in the 1970s C environment; things have changed and we can afford to be a little bit more verbose for clarity.
- ↗ It's not necessarily the most efficient, and certainly not the only way:
  - ↗ <http://codinghighway.com/2013/07/13/the-magic-of-wrappers/>
  - ↗ [http://www.inf.udec.cl/~leo/Malloc\\_tutorial.pdf](http://www.inf.udec.cl/~leo/Malloc_tutorial.pdf)
  - ↗ <https://danluu.com/malloc-tutorial/>

# How K&R Malloc Works

- ↗ Linked list to track free memory

- ↗ Located in the Heap
- ↗ *Circular Linked List*



- ↗ This is the *Free List*

- ↗ Contains memory available to be malloc'ed
- ↗ Once memory is malloc'ed, we don't track it until it is free'd

# How K&R Malloc Works

- `void *malloc(size_t n);`
  - Delete a node from the free list
  - Return a pointer to that node to the program
- `void free(void *ptr)`
  - Insert the node ptr into the free list
- Note: We've renamed them `_malloc()` and `_free()` in our code to keep from conflicting with the `malloc()` and `free()` from the C library
  - There are a number of C library functions that use `malloc()` and `free()`; we don't want override the library versions until our versions work properly!

# Question

In the K&R implementation of malloc(), the data structure in the heap is a:

- A. Binary Tree
- B. Hash Table
- C. Circular Linked List
- D. Array of Structs



# Question

In the K&R implementation of the heap management, free() does the following:

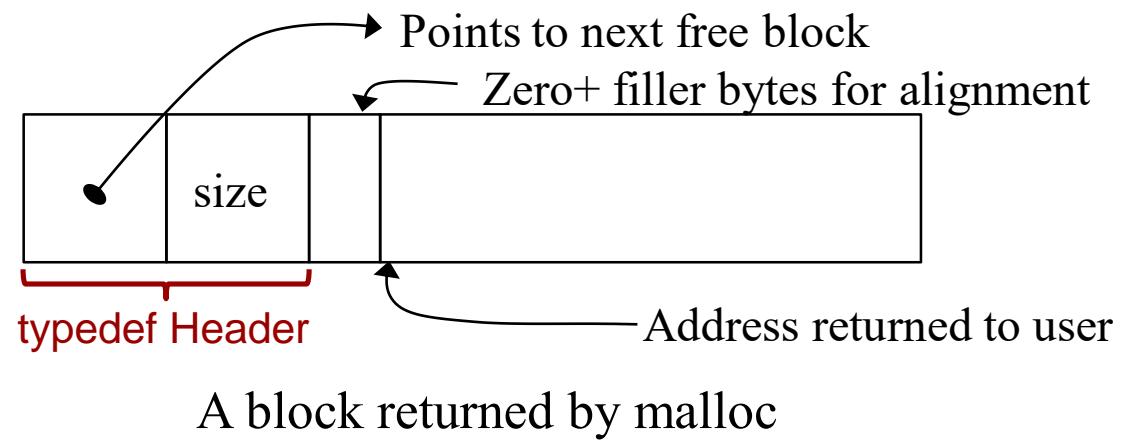
- A. Deletes a node from the free list
- B. Inserts a node in the free list
- C. Zeroes out the bytes on the heap
- D. Reduces the size of the heap



# How K&R Malloc Works

- Each node on the free list is available memory
  - Everything on the free list is in heap memory areas that malloc has requested from the system
  - Other functions can request heap space from the system, but that space will never show up on malloc's free list
- Consider this:
  - We want each block in the free list to be as large as possible
    - So it is more likely to have enough bytes to satisfy a malloc() call
    - Therefore when we free a block adjacent to our other memory on the heap, we should merge that block into the adjacent block.

# A Node in the Free List



# How K&R Malloc Works

## ↗ Design choices

- ↗ Linked List is ordered by memory address (in the heap)
- ↗ We could also order the linked list from largest to smallest size
- ↗ There are many other ways we could implement malloc()

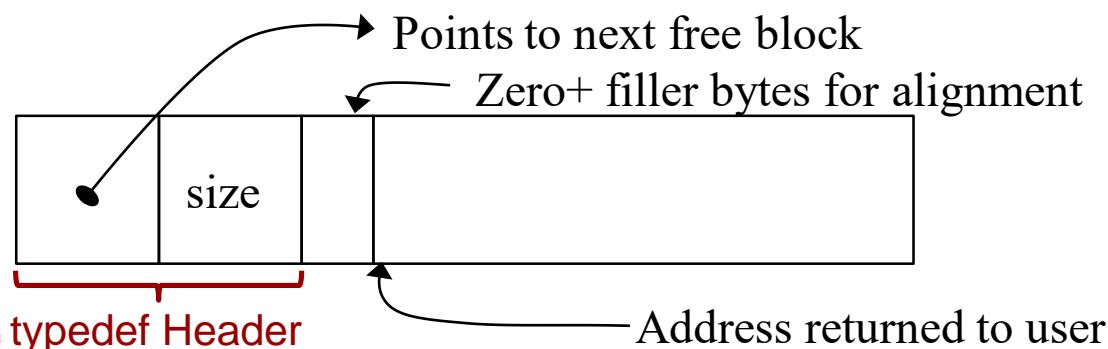
## ↗ Size in K&R malloc is number of *units*

- ↗ Not number of bytes
- ↗ A block is sizeof(Header) rounded up to the strictest alignment boundary
- ↗ A block is 16 bytes in this implementation

# Section of K&R Code

```
struct header { /* block header */
 struct header *next; /* next block if on free list */
 unsigned size; /* size of this block in Headersize units */
};
typedef struct header Header;

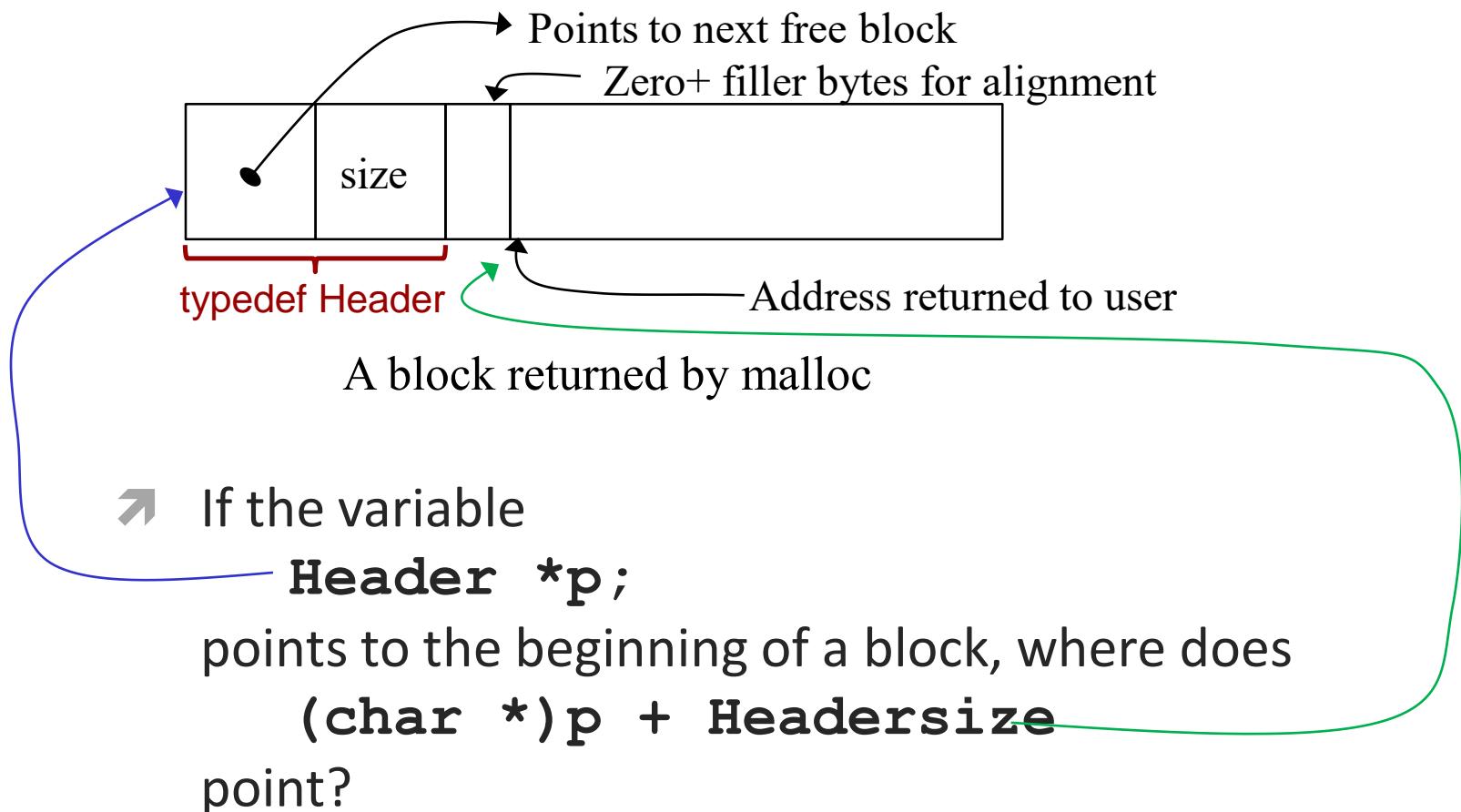
#define ALIGNBOUNDARY 16 /* align memory on 16 byte boundaries */
static int Headersize /* rounded up; use in place of sizeof(Header) */
 = ((sizeof(Header) - 1) / ALIGNBOUNDARY + 1) * ALIGNBOUNDARY;
```



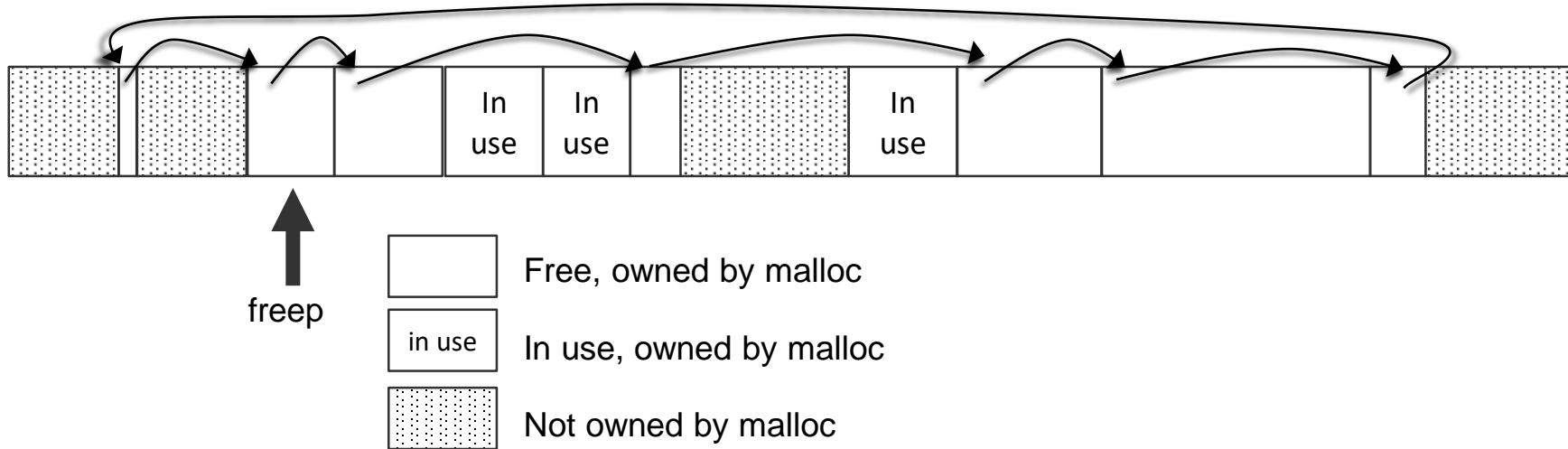
A block returned by malloc

| sizeof(Header) | Headersize |
|----------------|------------|
| 8              | 16         |
| 12             | 16         |
| 16             | 16         |
| 20             | 32         |

# Returned by Malloc



# Heap Layout



- ↗ K&R ch. 8
- ↗ One of those free blocks has user data size 0 and isn't in the heap, so it is never handed over by malloc().

# Section of K&R Code

```
static Header base; // empty list to get started with
static Header *freep = NULL; // start of free list

/* malloc: general-purpose storage allocator */
void *malloc(unsigned nbytes)
{
 Header *p, *prevp;
 unsigned long nunits;

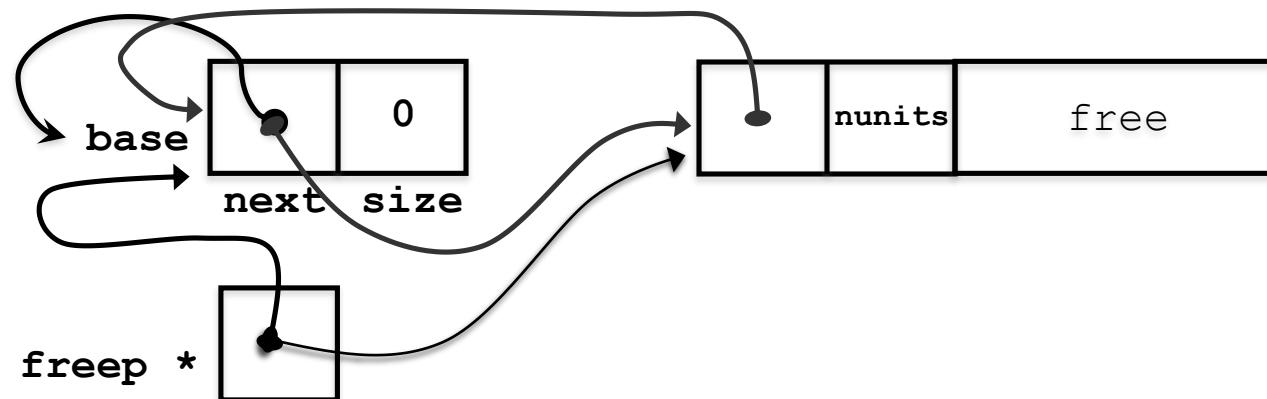
 /* Allocate memory in Headersize units, so round up request */
 nunits = (nbytes + Headersize - 1) / Headersize + 1;
```

- The key here is alignment with the Header's size
- You need (just) enough bytes to cover the program's request and the header information
- Suppose that the Header consists of 16 bytes [Headersize = 16]
- Take a look at different requests for sizes as represented by nbytes

| nbytes | 7 | 15 | 16 | 17 | 32 | 33 |
|--------|---|----|----|----|----|----|
| nunits | 2 | 2  | 2  | 3  | 3  | 4  |

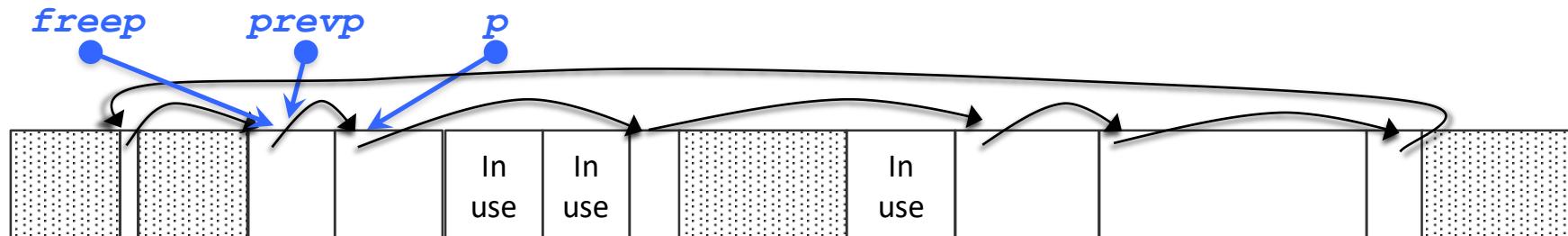
# Section of malloc() Code

```
if (freep == NULL) {
 freep = &base;
 base.size = 0;
 base.next = &base;
 if (morecore(nunits) == NULL)
 return NULL; /* none left; give up */
}
```



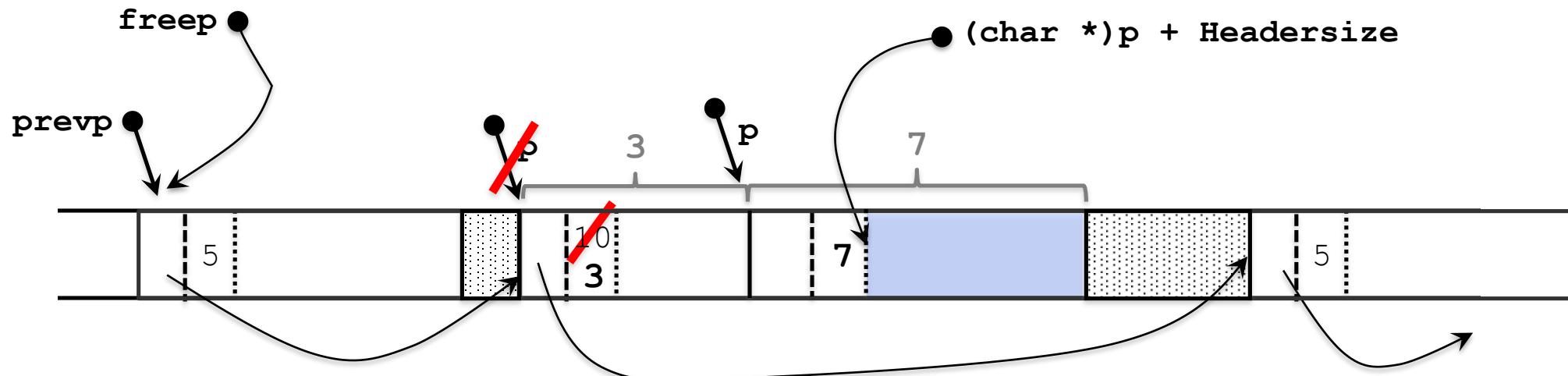
# Section of malloc() Code

```
for (prevp = freep, p = prevp->next; p->size < nunits;
 prevp = p, p = p->next) {
 if (p == freep)
 if ((p = morecore(nunits)) == NULL)
 return NULL; /* none left; give up */
}
```



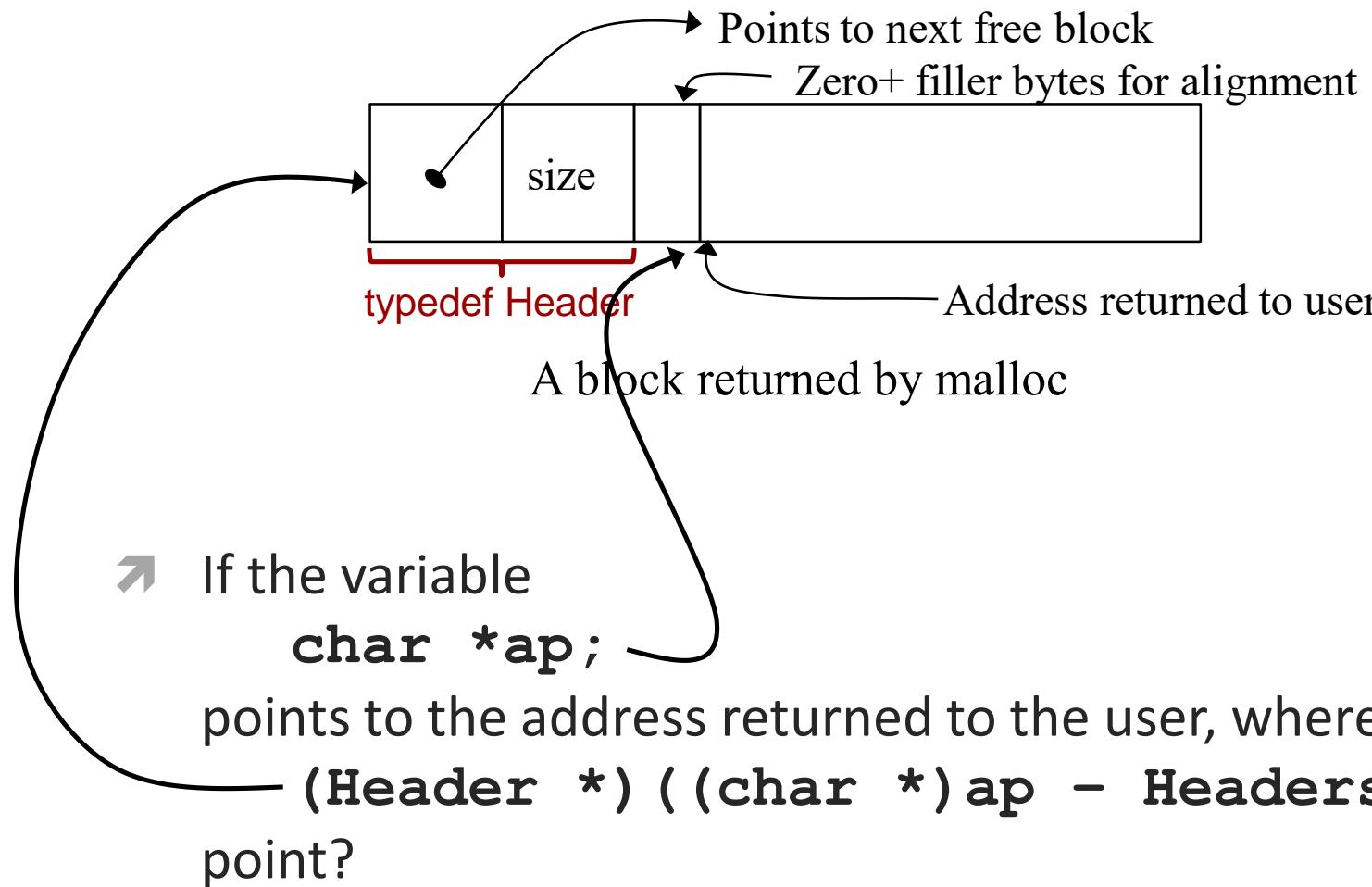
# Section of malloc() Code

```
if (p->size == nunits)
 prevp->next = p->next;
else {
 p->size -= nunits;
 p = (char *)p + p->size * Headersize;
 p->size = nunits;
}
freep = prevp;
return (char *)p + Headersize;
```



Let `nunits` = 7, and follow how the free block is identified and returned

# Pointer Sent to free()

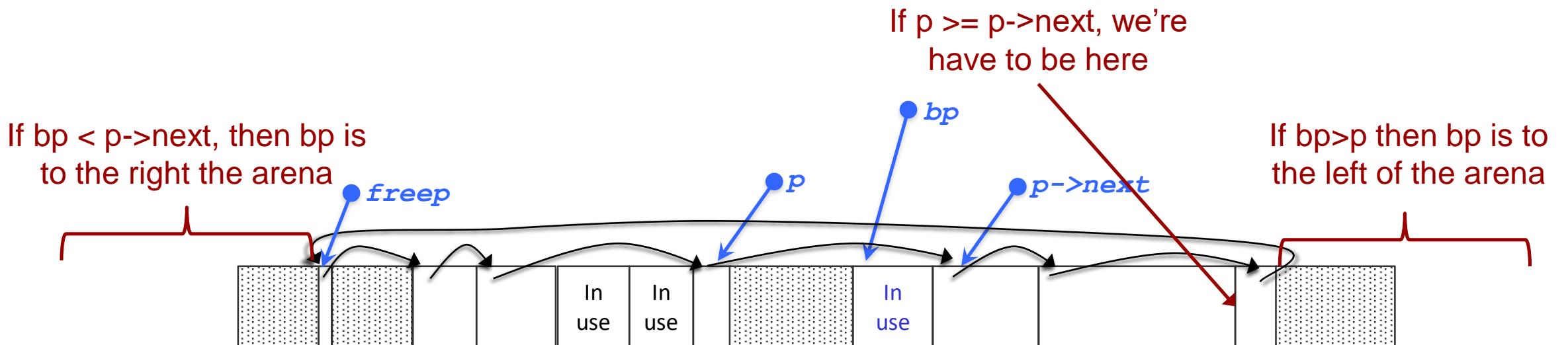


# Section of free() Code

```
void _free(void *ap)
{
 Header *bp, *p;

 bp = (Header *) (ap - Headersize);

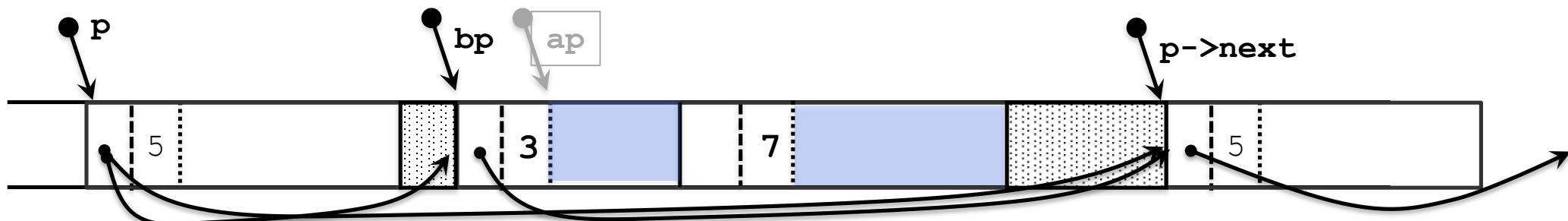
 /* Find p such that the freed block is between p and p->next */
 for (p = freep; !(p < bp && bp < p->next); p = p->next)
 /* is *bp at either end of the arena? */
 if (p >= p->next && (bp > p || bp < p->next))
 break;
```



# Free(): Below and Above In Use

```
/* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == (char *)p->next) {
 bp->size += p->next->size;
 bp->next = p->next->next;
} else
 bp->next = p->next;

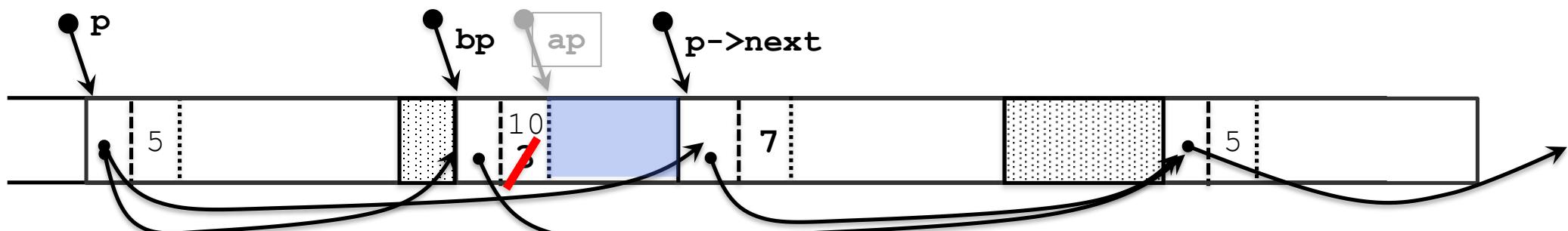
/* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == (char *)bp) {
 /* add the freed block to the block at p */
 p->size += bp->size;
 p->next = bp->next;
} else
 p->next = bp;
freep = p;
```



# Free(): Below In Use

```
/* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
 bp->size += p->next->size;
 bp->next = p->next->next;
} else
 bp->next = p->next;

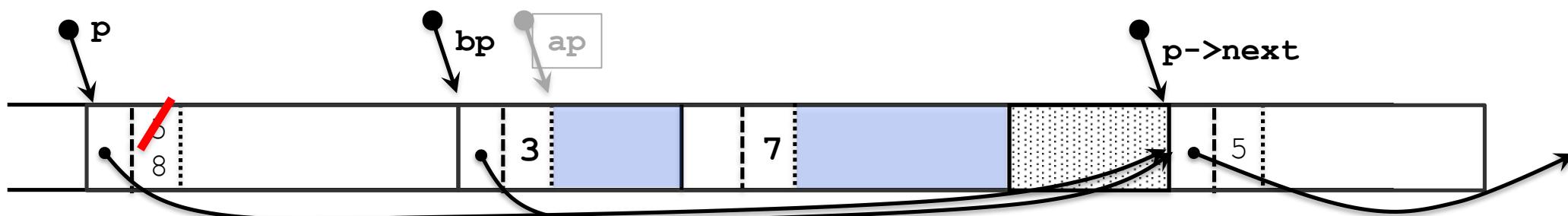
/* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
 /* add the freed block to the block at p */
 p->size += bp->size;
 p->next = bp->next;
} else
 p->next = bp;
freep = p;
```



# Free(): Above In Use

```
/* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
 bp->size += p->next->size;
 bp->next = p->next->next;
} else
 bp->next = p->next;

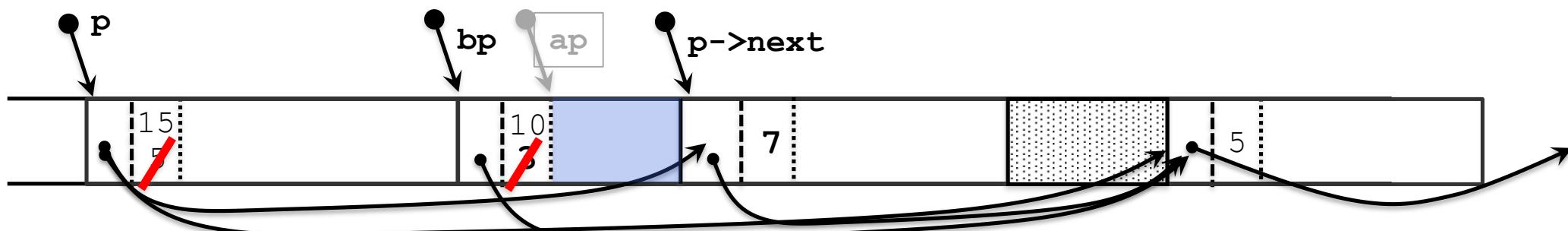
/* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
 /* add the freed block to the block at p */
 p->size += bp->size;
 p->next = bp->next;
} else
 p->next = bp;
freep = p;
```



# Free(): Above and Below Free

```
/* Look to see if we're adjacent to the block after the freed block */
if ((char *)bp + bp->size * Headersize == p->next) {
 bp->size += p->next->size;
 bp->next = p->next->next;
} else
 bp->next = p->next;

/* Look to see if we're adjacent to the block before the freed block */
if ((char *)p + p->size * Headersize == bp) {
 /* add the freed block to the block at p */
 p->size += bp->size;
 p->next = bp->next;
} else
 p->next = bp;
freep = p;
```

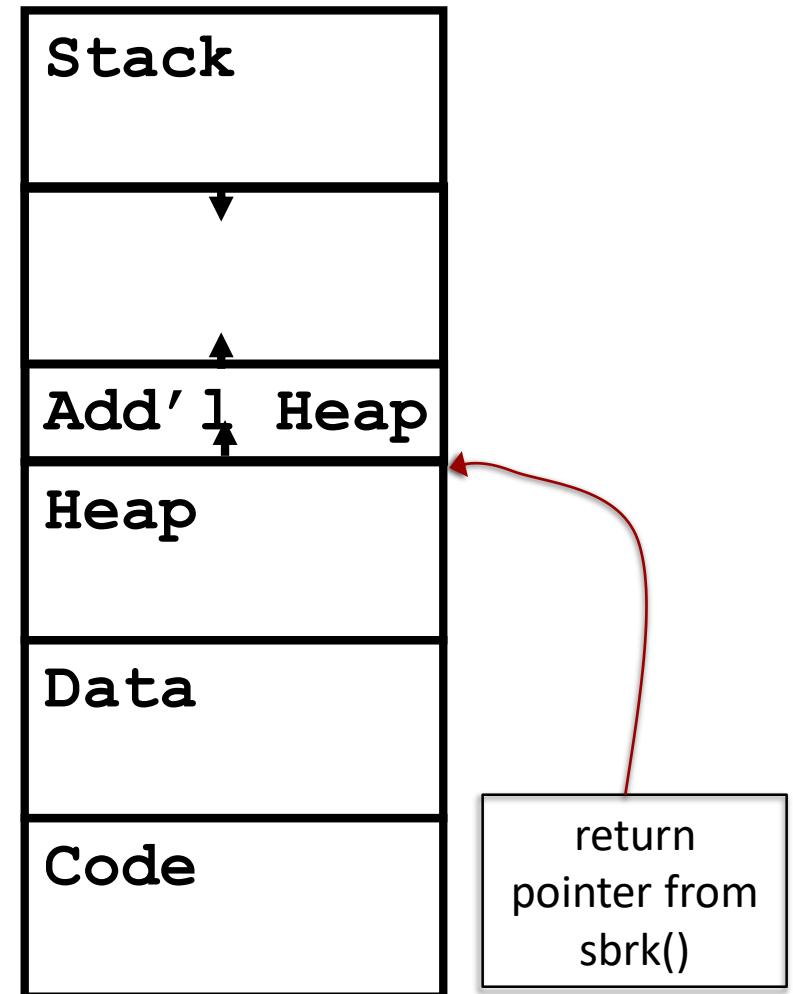


## Section of morecore() Code 2

```
/* Don't ask for any less than NALLOC */
if (nu < NALLOC)
 nu = NALLOC;
cp = sbrk(nu * Headersize);
if (cp == (char *) -1) /* no space */
 return NULL;

/* Take the new block, add a header, and free() it */
up = (Header *) cp;
up->size = nu;
_free((char *)up + Headersize);
return freep;
```

- `sbrk()` is a Linux system call to obtain more memory.
- It works at the end of the data segment/heap (a.k.a. “the break”) to acquire at least  $n$  bytes more space from the OS
- It returns an aligned pointer to the new space or -1 if no space is available
- The -1 is an odd historical artifact



# Question

After the execution of this code,

```
double *mp = &d[3];
char *xp = (char *)mp + 16;
```

to what memory address does *xp* point?

- A. 0x10 bytes past the address of *d*[3]



- B. 10 bytes past the beginning of *d*

- C. 10 bytes before the beginning of *d*

- D. 16 bytes before the beginning of *d*

Today's number is  
16,661

# Memory Management

- What are the equivalents of malloc and free in Java? Python?
  
- How can we “automatically” collect garbage?
  - Mark & Sweep Techniques
  - Reference Counting Techniques
  - And many, many others...

# Miscellaneous Topics



# Control Flow

- There is a goto in C but it is not recommended for several reasons
  - Optimization
  - Performance
  - Clarity
- Plus it will only jump within a single function
- Consider these two example code statements

```
while (a < 5) { | A: if (a >= 5) goto B;
```
- How does control reach each statement (as in what possible lines executed before each of these statements)?
  - For the while, control must come from the statement preceding the while() or it must come back from the end if the while loop
  - For the labelled statement, control can come from **anywhere** in the function! You must search for all the goto statements to see which ones reference A!

- ↗ Even worse...
- ↗ There is a C library construct that will allow jumping all over the place.
- ↗ **USE IT SPARINGLY!!!!**
- ↗ (This is the underpinning of the Java exception-throwing facility, exposed for all sorts of uses and abuses.)

# setjmp() ... longjmp()

```
#include <setjmp.h>
jmp_buf x; // this will hold location x
...
t = setjmp(x); // mark a location as x
 // t will be 0 for normal
 // ctrl flow, non-zero if
 // coming from longjmp()
...
longjmp(x, 3); // transfer back to location
 // x, and setjmp will have
 // the return value 3 (an
 // integer)
```

**Note: Read Wikipedia Article!!!!**

# setjmp() ... longjmp()

- A call to setjmp() saves various information about the calling environment (typically, the stack pointer, the program counter, possibly the values of other registers and the signal mask) in the jmp\_buf and returns 0
- longjmp() must be called later in the function OR in a function that's called by the function that did the setjmp(). In other words, the stack frame of the function that called setjmp() must still be on the stack (somewhere) when longjmp() is called
- Longjmp() restores the saved CPU state from jmp\_buf while setting the return value to the value in its second argument
- This makes it appear as if setjmp() returns for a SECOND time without being called again, distinguished only by its return value!

# Variadic Functions

- ↗ We have to have type-conformance of all of our arguments and parameters, right?
- ↗ So how the heck does printf() get away with calls like
  - ↗ `printf("Hello world!\n");`
  - ↗ `printf("Count %d, average %f\n", ct, avg);`
  - ↗ `printf("%d %d %d %d\n", a[0], a[3], a[2], a[1]);`

# Enter stdarg.h

```
#include <stdarg.h>
void printargs(int arg1, ...);

int main(void) {
 printargs(5, 2, 14, 84, 97, 15, -1, 48, -1);
 printargs(84, 51, -1);
 printargs(-1);
 printargs(1, -1);
 return 0;
}
```

# Printargs()

```
void printargs(int arg1, ...)
{
 va_list ap;
 int i;

 va_start(ap, arg1);
 for (i = arg1; i >= 0; i = va_arg(ap, int))
 printf("%d ", i);

 va_end(ap);
 putchar('\n');
}
```

# And the result...

```
$./a.out
5 2 14 84 97 15
84 51
```

```
1
$
```

```
int main(void)
{
 printargs(5, 2, 14, 84, 97,
 15, -1, 48, -1);
 printargs(84, 51, -1);
 printargs(-1);
 printargs(1, -1);
 return 0;
}
```

# Question

Which of these statements are true about C functions with variable numbers of arguments?

- A. C sets the first function argument to an integer count of the number of arguments passed
- B. There must always be at least one argument in the call and in the function prototype
- C. It is the responsibility of the called function to figure out how many arguments it was passed
- D. B and C
- E. All of the above



# The printf() prototype

- And by the way, the prototype for printf() is simply

```
void printf(char *format, ...);
```

- And note that printf() and its friends use the format string to determine how many arguments are present. Gcc even goes out of its way to check that for you.

# Main() Return Value

- 0 means okay
- Anything else means a problem
- Main() can return an 8-bit value
- Any function can exit and set return value

```
exit(99);
```

- You can see the value on the command line with:

```
echo $?
```

# Idioms for Opening Files

- In the C environment, there are three files opened for you on pre-defined “streams”, typically connected to your keyboard/display
  - stdin – Standard input
  - stdout – Standard output – printf(...) defaults to stdout
  - stderr – Standard error output – use fprintf(stderr, ...)
- Any other files must be opened and closed by calling a Standard IO routines

```
FILE *infile;
if((infile = fopen("f.txt","r")) == NULL) {
 // Handle error
}

FILE *infile;
if(!(infile = fopen("f.txt","r"))) {
 // Handle error
}
```

# Unformatted (Binary) I/O

```
#include <stdio.h>

size_t fread(void *ptr, // Read size*nmemb bytes
 size_t size, // from stream
 size_t nmemb,
 FILE *stream);

size_t fwrite(const void *ptr, // Write size*nmemb bytes
 size_t size, // to stream
 size_t nmemb,
 FILE *stream);
```

# Buffered Output

- The Standard IO library provides buffering improvements to minimize the number of system call traps that need to be made to read() and write()
- For stdout the output is line-buffered if it's written to a terminal and block-buffered if it's written to a disk file
  - If your program crashes, the buffer contents may be lost, especially if the file has been redirected to a disk file instead of a terminal.
  - That means you can lose your debugging output from printf if you don't end it with a newline and don't allow it to go directly to your terminal
- If you execute these printf statements just before a seg fault, you may very well not see any output(!)

```
printf("Checkpoint 1..");
```

...

```
printf("Checkpoint 2..");
```

# Buffered Output

- You can force the output buffer to be flushed with `fflush()`
- If the program crashes, the buffer contents may be lost

```
printf("Checkpoint 1");
fflush(stdout);
```

...

```
printf("Checkpoint 2");
fflush(stdout);
```

- You can also set the buffering with a call to a function in the `setvbuf()` family

# NOTE!

- For stderr the output is set to NOT buffered
- So output to stderr will always cause immediate I/O

```
fprintf(stderr, "Checkpoint 1");
```

```
fprintf(stderr, "Checkpoint 2");
```

## Question

The main reason that output is buffered by default in Standard IO before sending the output to the operating system is

- A. To minimize the use of operating system buffers
- B. To reduce the number of I/O traps the operating system has to process
- C. To avoid using the complicated read() and write() system calls
- D. To reduce the memory footprint of I/O operations



# Functions may be Inlined

```
inline int myfunc(int a, int b) {
 return a+b;
}
```

# Optimization

- ☞ You use the ***-Olevel*** option to control optimization
  - ☞ 0 – don't move code around (the default)
  - ☞ 1 or just **-O** – quick optimizations that don't take much compilation time
  - ☞ 2 – all optimizations that don't involve speed/space tradeoff
  - ☞ 3 – even more optimization
- ☞ The higher the optimization level, the more likely your code will not work properly if you don't follow the C rules precisely!
- ☞ For use with a debugger, use **-O0** or **-O1**.

*"We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%"*

-- Donald Knuth

- ↗ Don't write tricky code designed to be very efficient
  - ↗ You'll probably just defeat the optimizer by hiding what you're trying to accomplish
  - ↗ Until they've measured it, programmers *rarely* know where the CPU is spending its time in their code; this is called *premature optimization*
  - ↗ If you're really trying to be efficient in these modern times...
    - ↗ (1) measure, a.k.a. *profile*, your program to see where it spends its time
    - ↗ (2) look at the optimized assembly language output (**-S**) for your hot spots to see how the code is being generated
    - ↗ (3) Modify your C code to improve the optimized output
    - ↗ (4) Measure it again!

- Speaking of tricky, what does this mean?

`_ += ++_ + _--;`

- Can you identify a problems?
  - `++` and `--` have side effects – it matters in what order they are executed
  - And the C standard doesn't specify an ordering for operations in an expression except in specific cases (`&&`, `||`, `,`)

# What Does This Program Do?

```
#define P(a,b,c) a##b##c
#include/******<curses.h>
int c,h, v,x,y,s, i,b; int
main () { initscr(); P(cb,
rea, k()) ;/// ho)(
P(n, oec, curs_set(0); s= x=COLS/2
)/* */ ;for shi, np(){ timeout(y=c= v=0);///
; P(flu, r()) ;for (P(
P(c, lea, dstr) (2, 3+x,
mva, d, dstr usl, eep,)(U)){//
G) ; ; P(vad, dstr)(y >>8,x,//
P(m, "); for(i=LINES; /* * */
; mvinsch(i,0,0>(~c|i-h-H if((i- h|h- i+H) <0?'|' :'='));
:(i- i=(y +v= getch()>0?I:v+
A)>>8)>=LINES| |mvinch(i*= 0<i, x)!=' '| '|'
!=mvinch(i,3+x)break/*&% &*/; mvaddstr(y
>>8, x,0>v ?F:B); i--s
/-W; P(m, vpr, intw)(0,
COLS-9," %u/%u ",(0<i)* b); refresh(); if(++ i,b=b<i?i:c
b); refresh(); if(++ c==D){ c
--W; h=rand()% (LINES-H-6
)+2; } } flash(); }}
```

# An ASCII Version of Flappy Birds

↗ <https://www.youtube.com/watch?v=ReWzbwevcuY>

# International Obfuscated C Code Contest

↗ <http://www.ioccc.org>

Profiling a program means

- A. Looking to see where a program is fattest so it can be slimmed down in those regions
- B. Instrumenting and measuring a running program to discover where the CPUs spend their time in order to find the most effective places to optimize the code
- C. Optimizing a program by shaving off fuzzy portions of ] its data structures
- D. The inequitable practice of judging a program's correctness by relying heavily on its external appearance



Today's number is 14,285



# Questions?

# Signals

- <http://www.csl.mtu.edu/cs4411.choi/www/Resource/signal.pdf>
- The following slides are from
  - [www.cs.fsu.edu/~xyuan/co3p4610/lecture\\_7\\_osinterface5.ppt](http://www.cs.fsu.edu/~xyuan/co3p4610/lecture_7_osinterface5.ppt)

- IPC mechanism: Signal
  - Tells a process that some event occurs. It occurs
    - In the kill command.
      - Try ‘kill -l’
      - ‘kill -s INT #####(pid)’
    - When Ctrl-C is typed (SIGINT).
    - When Ctrl-\ is typed (SIGQUIT)
    - When a child exits (SIGCHLD to parent)
    - When a timer expires
    - When a CPU execution error occurs
    - .....
  - A form of inter-process communication.

# Available Actions and Signals

- When a process receives a signal, it performs one of the following three options:
  - Ignore the signal
  - Perform the default operation
  - Catch the signal (perform a user defined operation).
- Some commonly used signals:
  - SIGABRT, SIGALRM, SIGCHLD, SIGHUP, SIGINT, SIGUSR1, SIGUSR2, SIGTERM, SIGKILL, SIGSTOP, SIGSEGV, SIGILL
  - All defined in signal.h

# Processing Signals

- Processing signals:
  - similar to an interrupt (software interrupt)
  - when a process receives a signal:
    - pause execution
    - call the signal handler routine
    - Continue execution
  - Signal can be received at any point in the program.
  - Most default signal handlers will terminate the program.
  - You can change the way your program responses to signals.
    - E.g Make Ctrl-C have no effect.

# Simplified Signal Interface

- ANSI C signal function to change the signal handler
  - syntax:
    - #include <signal.h>
    - void (\*signal(int sig, void (\*disp)(int)))(int);
    - Alternately
      - typedef void (\*sighandler)(int);
      - sighandler signal(int sig, sighandler disp);
  - semantics:
    - sig -- signal (defined in signal.h)
    - disp: SIG\_IGN, SIG\_DFL or the address of a signal handler.
    - Handler may be reset to SIG\_DFL after one invocation
      - AT&T UNIX does the reset, BSD UNIX does not do the reset
      - Using signal with a handler function isn't portable; use sigaction(2)
      - How to get continuous coverage?
      - Still have problems – may lose signals

# Question

A call to *signal()* in C

- A. Carries a payload of 128 bytes
- B. Always causes the process to terminate
- C. Establishes the behavior for the process when it receives a particular signal
- D. Catches the signal for later use by the C Standard Library



Today's number is 54,321

# A Non-Portable Example Using signal(2)

```
#include <signal.h>

void sigcatcher(int);
void sigexiter(int);

int main(int argc, char *argv[]) {
 signal(SIGINT, sigcatcher); // control-c
 signal(SIGQUIT, sigcatcher); // control-\
 signal(SIGTERM, sigexiter); // "kill process-id"

 printf("My process id is %d\n", getpid());
 while (1) {
 printf("Waiting 30 seconds on a signal\n");
 sleep(30);
 }
}

void sigcatcher(int s) {
 signal(s, sigcatcher);
 printf("Caught signal %d\n", s);
}

void sigexiter(int s) {
 printf("Exiting on signal %d\n", s);
 exit(1);
}
```

# At Least Two Things Wrong

```
#include <signal.h>

void sigcatcher(int);
void sigexiter(int);

int main(int argc, char *argv[]) {
 signal(SIGINT, sigcatcher); // control-c
 signal(SIGQUIT, sigcatcher); // control-\
 signal(SIGTERM, sigexiter); // "kill process-id"

 printf("My process id is %d\n", getpid());
 while (1) {
 printf("Waiting 30 seconds on a signal\n");
 sleep(30);
 }
}

void sigcatcher(int s) {
 signal(s, sigcatcher);
 printf("Caught signal %d\n", s);
}

void sigexiter(int s) {
 printf("Exiting on signal %d\n", s);
 exit(1);
}
```

Printf() isn't  
"signal safe" and  
can cause a  
deadlock in a  
signal handler

write() is OK  
though

This resets the signal  
handler if the OS is one  
that resets it to default

There is a race condition  
if another signal comes  
in just before this call –  
the signal will cause the  
default action instead of  
calling the handler

# Blocking Temporarily Suspends Signal Actions

- Block/unblock signals

- Manipulate signal sets

- #include <signal.h>

- int sigemptyset(sigset\_t \*set);

- int sigfillset(sigset\_t \*set);

- int sigaddset(sigset\_t \*set, int signo);

- int sigdelset(sigset\_t \*set, int signo);

- int sigismember(const sigset\_t \*set, int signo);

- Manipulate signal mask of a process

- int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oset);

- How: SIG\_BLOCK, SIG\_UNBLOCK, SIG\_SETMASK

# Example of Deferring a Signal

- For a critical region where you don't want a certain signal to be deferred, the program will look like:

```
#include <signal.h>
sigset_t newmask, oldmask;
sigemptyset(newmask);
sigaddset(newmask, SIGINT);

sigprocmask(SIG_BLOCK, &newmask, &oldmask);
..... /* critical region */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

## ↗ sigaction

- ↗ Supersedes the signal function
- ↗ #include <signal.h>
- ↗ int sigaction(int signo, const struct sigaction \*act, struct sigaction \*oact)  
struct sigaction {  
 void (\*sa\_handler)(); /\* signal handler \*/  
 sigset\_t sa\_mask; /\*additional signal to be block \*/  
 int sa\_flags; /\* various options for handling signal \*/  
};

- Kill:

- Send a signal to a process
- `#include <signal.h>`
- `#include <sys/types.h>`
- `int kill(pid_t pid, int signo);`
  - Pid > 0, normal
  - Pid == 0, all processes whose group ID is the current process' group ID.
  - Pid <0, all processes whose group ID = |pid|

Which of the following is NOT a characteristic of C signals?

- A. Signals can be deferred until the program is ready to process them
- B. A process can send a signal to another process or to itself
- C. The receipt of a signal by a program can be ignored, can be processed according to a default rule, or can cause a function to be called
- D. Signals are only processed at the entry and exits of C functions



# Signalling Example: Counting Child Processes

- ↗ Signalling example: keeping track of number of child processes in a shell: when a process exits, it sends a **SIGCHLD** to its parent.

# Example: First Half

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>

int numofchild = 0;

Signal catcher { // If we catch a SIGCHLD, decrement the number of active children
void sigchildhandler() {
 numofchild--;
 write(1, "Child exited\n", sizeof("Child exited\n"));
}

int main() {
 char cmd[1000], buf[1000], *argv[2];
 struct sigaction abc;
 int pid;

Install the signal catcher { // Set a sigchildhandler() to catch SIGCHLD signals
abc.sa_handler = sigchildhandler;
sigemptyset(&abc.sa_mask);
abc.sa_flags = 0;
sigaction(SIGCHLD, &abc, NULL);
```

# Example: Second Half

```
while(1) {
 // Read in a command name to execute
 printf("<%d>", numofchild);
 fflush(stdout);
 while(fgets(buf, 100, stdin) == NULL)
 ;
 sscanf(buf, "%s", cmd);

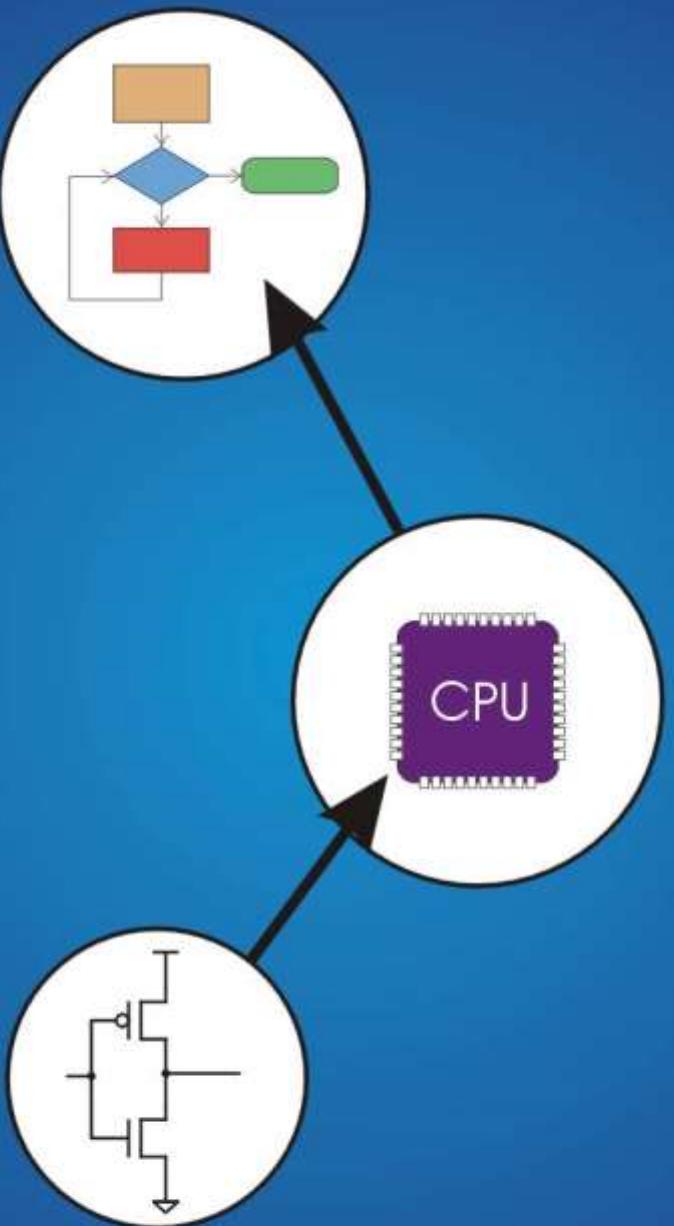
 // Command is "quit"; exit if all children are complete
 if (strcmp(cmd, "quit") == 0) {
 if (numofchild == 0)
 exit(0);
 printf("There are still %d children.\n", numofchild);
 } else if ((pid = fork()) == 0) {
 argv[0] = cmd;
 argv[1] = NULL;
 execv(argv[0], argv);
 exit(0);
 } // If fork() doesn't fail, increment the number of children
 } else if (pid != -1)
 numofchild++;
}
} /* example6.c */
```

Read an input command

If “quit”, exit if no children active

Fork and execute a child

If the fork succeeded increment the child count



# The ARM ISA for LC-3 Programmers

Copyright © Thomas M. Conte  
with sources from ARM, Ltd.  
Revised William D. Leahy Jr.

**Today's number is 72,000**

**The claim we made was...**

**LC-3 is easy to learn**

**We will teach you LC-3, and *then it will be “easy” to learn another, real ISA***

**Time to show you we were right!**

**A thought: A *new ISA is like a new car. The pedals are there, there’s a steering wheel, etc. But the headlight control, high beams, wipers, etc, are all in a slightly different place. Still, you can drive it without too much trouble, and not (hopefully) crash into something!***

# REVIEW: ISA= Instruction Set Architecture

**ISA** = The binary “language” used to talk to a processor. All of the *programmer-visible* components and operations of the computer

- **memory organization**
  - address space -- how many locations can be addressed?
  - addressability -- how many bits per location?
- **register set**
  - how many? what size? how are they used?
- **instruction set**
  - Operation codes)
  - data types (2's complement, floating-point, etc)
  - addressing modes.

## Some ARMisms

ARM was a small company in the UK. It was sold to SoftBank in 2016 and is being bought now by NVIDIA.

Technically, they make nothing – they sell the rights to use ARM-compliant processors in your design!

Originally called Acorn Computer Company

The Acorn was a small, cheap computer that used the Mostek 6502

Acorn decided to make their own processor: the Acorn RISC Machine (A.R.M.)

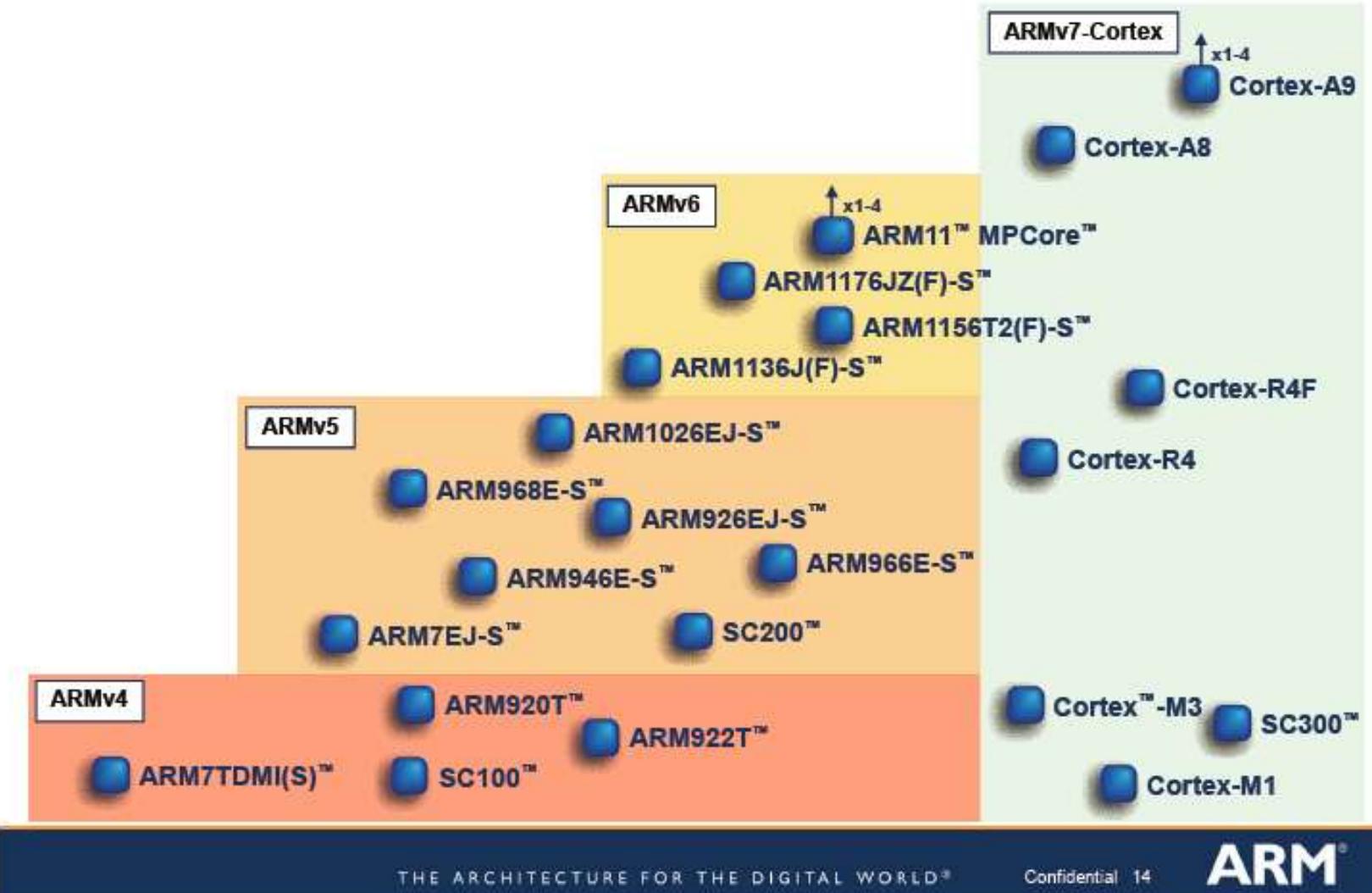
Since that time, A has become “Advanced” instead of “Acorn,” and company changed its name to “ARM”

ARM has had multiple *versions*

The *version* != the processor number

So ARM-7 does NOT use ARM-version7’s ISA (it uses ARM version 4)

# Architecture Versions



## What's the same as LC-3?

ARM and LC-3 are both “load/store” architectures

This means that values get into registers using a LOAD instruction and go to memory using a STORE instruction

ARM and LC-3 both have fixed-format instruction encodings

All instructions in LC-3 are 16b long. ARM has two ISAs, in one, they are 32b long (what I'll present), and they are 16b in the other

Register-register addressing

In ARM as in LC-3, many (most) instructions take two source registers and store their results in a destination register

## **Address space, addressability**

**LC-3 Address Space is  $2^{16}$  words, where a word is 16b**  
**all locations are word addressable**  
**the address size is 16 bits**

**ARM Address Space is  $2^{32}$  bytes**  
**all locations are byte-addressable**  
**the address size is 32 bits**

# **Processor Modes\***

|                   |                                               |
|-------------------|-----------------------------------------------|
| <b>User</b>       | <b>Normal execution mode</b>                  |
| <b>FIQ</b>        | <b>High priority (fast) interrupt request</b> |
| <b>IRQ</b>        | <b>General purpose interrupts</b>             |
| <b>Supervisor</b> | <b>Protected mode for O/S</b>                 |
| <b>Abort</b>      | <b>Used for memory access violations</b>      |
| <b>Undefined</b>  | <b>Used to handle undefined instructions</b>  |
| <b>System</b>     | <b>Runs privileged O/S tasks</b>              |

**LC-3 just has User and Supervisor modes**

# The ALU and the registers

ARM has 16 registers, R0-R15

UNLIKE LC-3, *the PC is a register that the programmer can access directly* (PC = R15 in ARM)

All registers are 32b wide

On older ARM ISAs (versions 1-5), storing to PC has “unpredictable results”

Other special purpose registers:

R11 (“FP”) is the frame pointer (similar function to R5 in LC-3)

R14 (“LR”) is the link register (same function as R7 in LC-3, it holds the return address for a subroutine call)

R13 (“SP”) is the stack pointer (same function as R6 in LC-3, it holds the address of the top of stack)

# The ARM Register Set: Register Banking

## Current Visible Registers

User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

## Banked out Registers

FIQ

r8

r9

r10

r11

r12

r13 (sp)

r14 (lr)

IRQ

r13 (sp)

r14 (lr)

SVC

r13 (sp)

r14 (lr)

Undef

r13 (sp)

r14 (lr)

Abort

r13 (sp)

r14 (lr)

spsr

spsr

spsr

spsr

spsr

# The ARM Register Set

## Current Visible Registers

FIQ Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr  
spsr

User

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

Banked out Registers

IRQ

r13 (sp)  
r14 (lr)

SVC

r13 (sp)  
r14 (lr)

Undef

r13 (sp)  
r14 (lr)

Abort

r13 (sp)  
r14 (lr)

spsr

spsr

spsr

spsr

# The ARM Register Set

## Current Visible Registers

User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

Banked out Registers

FIQ

IRQ

SVC

Undef

Abort

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

spsr

spsr

spsr

spsr

spsr

# The ARM Register Set

## Current Visible Registers

IRQ Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
**r13 (sp)**  
**r14 (lr)**  
**r15 (pc)**

**cpsr**  
**spsr**

User

**r13 (sp)**  
**r14 (lr)**

FIQ

**r8**  
**r9**  
**r10**  
**r11**  
**r12**  
**r13 (sp)**  
**r14 (lr)**

Banked out Registers

SVC

**r13 (sp)**  
**r14 (lr)**

Undef

**r13 (sp)**  
**r14 (lr)**

Abort

**r13 (sp)**  
**r14 (lr)**

**spsr**

**spsr**

**spsr**

**spsr**

# The ARM Register Set

## Current Visible Registers

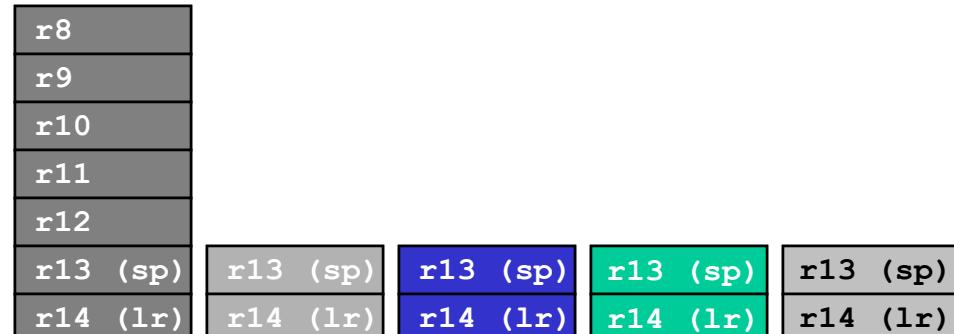
User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

Banked out Registers

FIQ      IRQ      SVC      Undef      Abort



spsr      spsr      spsr      spsr      spsr

# The ARM Register Set

## Current Visible Registers

SVC Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
**r13 (sp)**  
**r14 (lr)**  
**r15 (pc)**

cpsr  
**spsr**

User

**r13 (sp)**  
**r14 (lr)**

Banked out Registers

r8  
r9  
r10  
r11  
r12

**r13 (sp)**  
**r14 (lr)**

IRQ

Undef Abort

**r13 (sp)** **r13 (sp)**  
**r14 (lr)** **r14 (lr)**

**spsr** **spsr**

# The ARM Register Set

## Current Visible Registers

User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

Banked out Registers

FIQ

IRQ

SVC

Undef

Abort

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

spsr

spsr

spsr

spsr

spsr

# The ARM Register Set

## Current Visible Registers

Undef Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
**r13 (sp)**  
**r14 (lr)**  
**r15 (pc)**

**cpsr**  
**spsr**

User

Banked out Registers

r8  
r9  
r10  
r11  
r12

**r13 (sp)**  
**r14 (lr)**

**spsr**

**spsr**

**spsr**

**spsr**

SVC

Abort

# The ARM Register Set

## Current Visible Registers

User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

Banked out Registers

FIQ

IRQ

SVC

Undef

Abort

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

spsr

spsr

spsr

spsr

spsr

# The ARM Register Set

## Current Visible Registers

**Abort Mode**

|                 |
|-----------------|
| r0              |
| r1              |
| r2              |
| r3              |
| r4              |
| r5              |
| r6              |
| r7              |
| r8              |
| r9              |
| r10             |
| r11             |
| r12             |
| <b>r13 (sp)</b> |
| <b>r14 (lr)</b> |
| <b>r15 (pc)</b> |

|      |
|------|
| cpsr |
| spsr |

## Banked out Registers

**User**

|                 |
|-----------------|
| <b>r13 (sp)</b> |
| <b>r14 (lr)</b> |

**FIQ**

|            |
|------------|
| <b>r8</b>  |
| <b>r9</b>  |
| <b>r10</b> |
| <b>r11</b> |
| <b>r12</b> |

**IRQ**

|                 |
|-----------------|
| <b>r13 (sp)</b> |
| <b>r14 (lr)</b> |

**SVC**

|                 |
|-----------------|
| <b>r13 (sp)</b> |
| <b>r14 (lr)</b> |

**Undef**

|                 |
|-----------------|
| <b>r13 (sp)</b> |
| <b>r14 (lr)</b> |

 spsr | spsr | spsr | spsr |

# The ARM Register Set

## Current Visible Registers

User Mode

r0  
r1  
r2  
r3  
r4  
r5  
r6  
r7  
r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)  
r15 (pc)

cpsr

Banked out Registers

FIQ

IRQ

SVC

Undef

Abort

r8  
r9  
r10  
r11  
r12  
r13 (sp)  
r14 (lr)

spsr

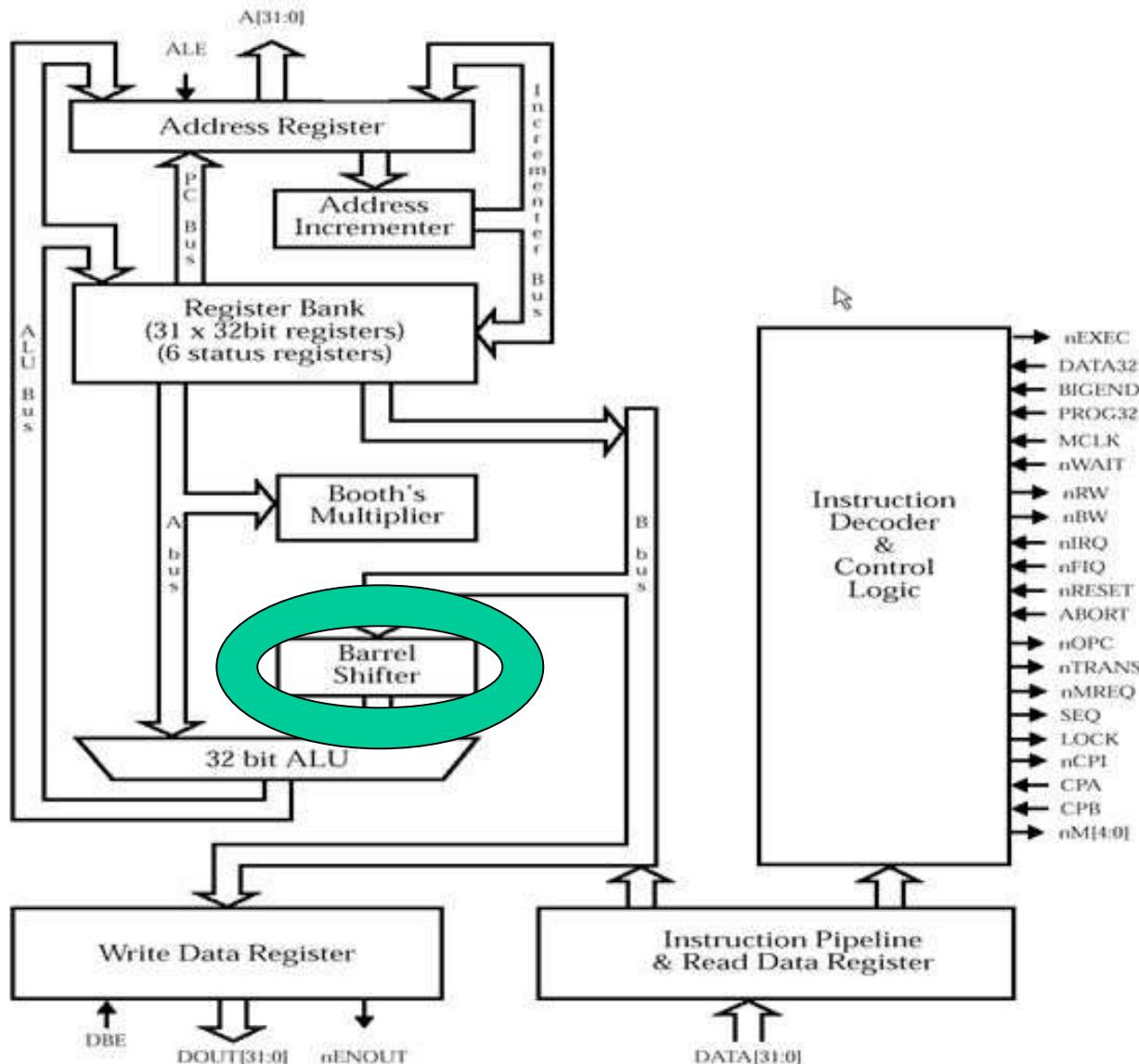
spsr

spsr

spsr

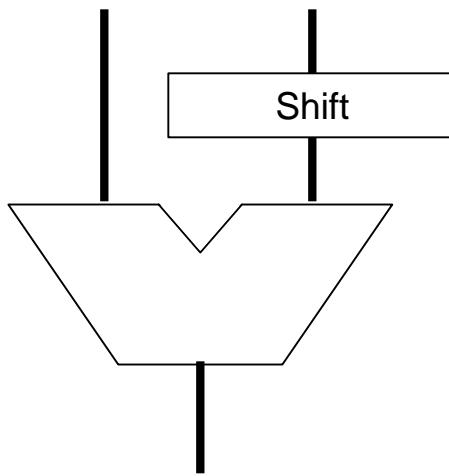
spsr

# ARMv7 Data Path



## Other “goodies”

The ALU in ARM has a “free shift” in front of it:



Logical shift left (lsl)  
Logical shift right (lsr)  
Arithmetic shift right (asr)  
Rotate right (ror)

## Conditional execution

In LC-3, the BR instruction uses the condition code register NZP

In ARM, the condition code register is called the CPSR (Current Program Status Register). It has:

N - Negative Result from ALU

Z - Zero result from ALU

C - ALU operation had a carry-out of MSbit

V - ALU operation had a twos-complement overflow  
(plus bits to handle interrupts, etc.)

In ARM, ALL instructions can “test” these bits

# Branching...

B label      or      BL label (branch and link)

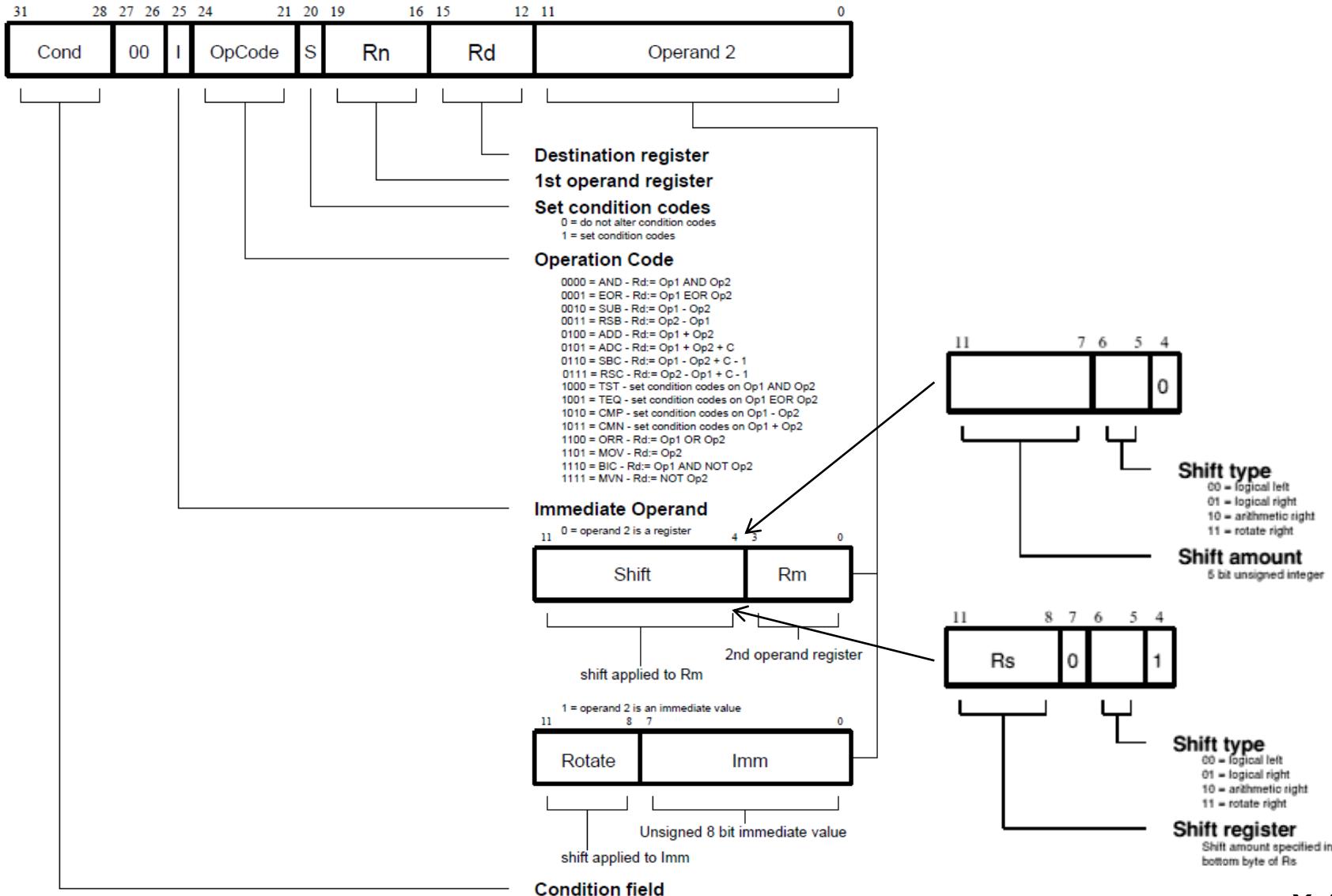
Conditional branching Bsuffix label...

| Suffix   | Flags   | Meaning                           |
|----------|---------|-----------------------------------|
| EQ       | Z set   | Equal                             |
| NE       | Z clear | Not equal                         |
| CS or HS | C set   | Higher or same (unsigned $\geq$ ) |
| CC or LO | C clear | Lower (unsigned $<$ )             |
| MI       | N set   | Negative                          |
| PL       | N clear | Positive or zero                  |

## more...

| Suffix | Flags                     | Meaning                                  |
|--------|---------------------------|------------------------------------------|
| VS     | V set                     | Overflow                                 |
| VC     | V clear                   | No overflow                              |
| HI     | C set and Z clear         | Higher (unsigned >)                      |
| LS     | C clear or Z set          | Lower or same (unsigned <=)              |
| GE     | N and V the same          | Signed >=                                |
| LT     | N and V differ            | Signed <                                 |
| GT     | Z clear, N and V the same | Signed >                                 |
| LE     | Z set, N and V differ     | Signed <=                                |
| AL     | Any                       | Always. This suffix is normally omitted. |

# ARM data processing instruction format



# Putting it all together: the ARM ADD instruction

ADD{S}<C> <Rd>, <Rn>, <Rm>, <type> <Rs>

|      |    |    |    |    |    |    |    |    |    |    |    |    |      |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |  |
|------|----|----|----|----|----|----|----|----|----|----|----|----|------|----|----|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|--|
| 31   | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18   | 17 | 16 | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |  |
| cond | 0  | 0  | 0  | 0  | 1  | 0  | 0  | S  | Rn | Rd | Rs | 0  | type | 1  | Rm |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |  |

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); s = UInt(Rs);
setflags = (S == '1'); shift_t = DecodeRegShift(type);
if d == 15 || n == 15 || m == 15 || s == 15 then UNPREDICTABLE;
```

Ignore “cond” for now

type = lsl, lsr, asr, or ror and applies to Rm

Examples:

ADD R1,R2,R3

r1 = r2+r3

ADDS R1,R2,R3

r1 = r2+r3 and the flags are set

ADDS R1,R2,R3 lsl R4

r1 = r2+(r3 lsl r4)

## Arithmetic opcodes that work like ADD

**op r1, r2, r3**

|            |                 |
|------------|-----------------|
| <b>SUB</b> | $r1 = r2 - r3$  |
| <b>RSB</b> | $r1 = r3 - r2$  |
| <b>AND</b> | $r1 = r1 \& r2$ |
| <b>ORR</b> | $r1 = r1   r2$  |
| <b>EOR</b> | $r1 = r1 ^ r2$  |

**Subtract**

**Reverse subtract**

**And**

**Or**

**Exclusive or**

**not a comprehensive list!**

**ARM has about 200 instructions depending on how you count!**

## Some two-operand opcodes

***op r1, r2***

**CMP** set flags based on r1-r2

**MOV** r1, r2

note can use an immediate if needed...

**CMP r1,r2**

**MOV r1,#0**

and shift...

**MOV r1,r2, lsl #2**

## Loads and stores

**LDR** *type*      R1, [R2, #imm]

**STR** *type*      R1, [R2, #imm]

What's *type*? Something LC-3 didn't have:  
**type is one of**

- B for byte (8b)
- H for halfword (16b)
- (blank) for word (32b)
- D for doubleword (64b)

Also can add an “S” for *sign extension*  
so you can have

**LDRSB**      R1, [R2,#4] ; Loads a byte into a 32-bit register and sign extends it

# All kinds of ways to get to memory...

[Rx, #imm]

[Rx, +Ry]      [Rx, -Ry]

[Rx,  $\pm$ Ry, *shift* #imm]

also add ! to update the Ry register afterwards

LDRD R1,[R2, -R3, lsl #3]!

effective address is  $R2 - (R3 \ll 3)$

after the load executes:  $R3 = R2 - (R3 \ll 3)$

or this syntax to update the register before the load

LDRD R1,R2,[-R3] lsl #3

$R3 = R2 - R3 \ll 3$  before the load executes

then the effective address is R3

# How do you get an address in a register?

Same conundrum as in LC-3...

In LC-3

`LEA R1,label`

In ARM

`LDR R1,=/label/` puts the value of “*label*” in R1

## Calls and returns

A call is a BL *label* for “branch and link”

LR (R14) gets the current PC+4

A return is via a BX *Rx*, or usually BX LR



## Quick example

Count the number of positive values of an array of ints,  
A[ ],there are 16 values to investigate...

|        |                           |                     |
|--------|---------------------------|---------------------|
|        | LDR R0 ,=A                | ;Address of A in R0 |
|        | MOV R1 ,#15               | ;Countdown value    |
|        | MOV R2 ,#0                | ;Initialize counter |
| Label: | LDR R3 , [R0 ,+R1 lsl #2] | ;Get an array val   |
|        | CMP R3 ,#0                | ;Test it            |
|        | BLT Noadd                 | ;Ignore neg vals    |
|        | ADD R2 ,R2 ,#1            | ;Increment count    |
| Noadd: | SUBS R1 ,R1 ,#1           | ;Dec counter        |
|        | BGE Label                 | ;More if not done   |

## Ok some weird (cool) stuff

ARM has conditional execution for (nearly) all instructions.

Those conditions (LT, LE, GT, ...) can be appended to an opcode that allows it

So *instead of this*

CMP R3,#0

BLT Noadd

ADD R2,R2,#1

Noadd: ...

*we could have done this*

CMP R3,#0

ADDGE R2,R2,#1

## Quick example

Count the number of positive values of an array, there are 16 values to investigate...

```
LDR R0 ,=A
MOV R1 ,#15
MOV R2 ,#0
Label: LDR R3 , [R0 ,+R1 ls1 #2]
 CMP R3 ,#0
 ADDGE R2 ,R2 ,#1 ; Increment if >= 0
 SUBS R1 ,R1 ,#1
 BGE Label
```

# Stacks!

# **Often load or store multiple...**

## LDMxx sp!, {register list}

**where xx is the kind of stack**

**FD - full (sp points to top of stack), descending (--sp to push)\*\***

## FA - full, ascending (sp++ to push)

also

## **ED – empty descending**

## **EA – empty ascending**

**\*\* what the ARM calling convention uses, same as LC-3**

## **Unix calling convention for ARM**

**r0-r3 are the argument and scratch registers;**

**r0-r1 are also the return value registers**

**r4-r8, r10-r11 are callee-save registers**

**r9 special use**

**r11 is the frame pointer**

**r12 is a temporary (caller-save)**

**r13 the link register**

**r14 the stack pointer**

**r15 the PC**

**A cross compiler you can get**

**For Mac OSX:**

**<http://sourceforge.net/projects/yagarto/>**

**free, pre-built binaries. Be sure to read the “read me first” before you install**

**then try**

**eabi-none-arm-gcc -S prog.c  
makes a prog.s assembly file**

## Using GCC cross compiler...

```
int a[16];
main() {
 int i, j;
 for (i = 15; i >= 0; i--)
 if (a[i] >= 0) j++;
 printf("%d\n", j);
}
```

```

.LC0: ascii "%d\n12\000"
.text
main: stmdf sp!, {fp, lr}
 add fp, sp, #4
 sub sp, sp, #8
 mov r3, #15
 str r3, [fp, #-8]
 b .L2
.L4: ldr r3, .L5
 ldr r2, [fp, #-8]
 ldr r3, [r3, r2, ls1 #2]
 cmp r3, #0
 blt .L3
 ldr r3, [fp, #-12]
 add r3, r3, #1
 str r3, [fp, #-12]
.L3: ldr r3, [fp, #-8]
 sub r3, r3, #1
 str r3, [fp, #-8]
.L2: ldr r3, [fp, #-8]
 cmp r3, #0
 bge .L4
 ldr r0, .L6
 ldr r1, [fp, #-12]
 bl printf
 sub sp, fp, #4
 ldmdf sp!, {fp, lr}
 bx lr
.L5: .word a
.L6 .word .LC0
,,
```

; save fp and lr (store multiple)  
; set up frame pointer ☺  
; int i, j //make space for two local vars  
; for (i = 15; ...  
;  
; //go to test part of for loop  
; if (a[i] >= 0) //load in base of 'a'  
; ; //get i  
; ; //get a[i]  
;  
; //branch less than over if part  
; ; j++ //if part: get j  
; ; //increment j  
; ; //update j.  
; ...i--) //reinitialize part of for  
; ; //decrement i  
; ; //update i  
; ...i >= 0;... // test part of for loop  
;  
; //back to loop body if i >= 0  
; printf("%d\n, j)//get the address of the format string for printf  
; ; //get the value of j  
; ; //branch and link (call) printf  
; ; pop the stack frame  
; ; restore registers  
; ; return  
; ; pointer to a[0]  
; ; pointer to literal "%d\n"

## Thumb

**ROM is precious on an embedded device**

**Can “save space” in ROM using *Thumb***

**Thumb is a proper subset of ARM ISA**

**Instructions are 16b wide, which reduces what you can do.**

**Here's a Thumb ADD encoding:**

## Encoding T1

ARMv4T, ARMv5T\*, ARMv6\*, ARMv7

ADDS <Rd>, <Rn>, <Rm>

Outside IT block.

ADD<c> <Rd>, <Rn>, <Rm>

Inside IT block.

|    |    |    |    |    |    |   |    |    |    |   |   |   |   |   |   |
|----|----|----|----|----|----|---|----|----|----|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8  | 7  | 6  | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 0  | 0 | Rm | Rn | Rd |   |   |   |   |   |   |

```
d = UInt(Rd); n = UInt(Rn); m = UInt(Rm); setflags = !InITBlock();
(shift_t, shift_n) = (SRTYPE_LSL, 0);
```

## Encoding T2

ARMv6T2, ARMv7 if <Rdn> and <Rm> are both from R0-R7

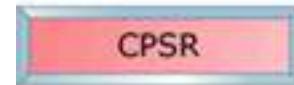
ARMv4T, ARMv5T\*, ARMv6\*, ARMv7 otherwise

ADD<c> <Rdn>, <Rm>

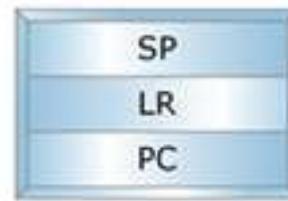
If <Rdn> is the PC, must be outside or last in IT block.

|    |    |    |    |    |    |   |   |    |    |     |   |   |   |   |   |
|----|----|----|----|----|----|---|---|----|----|-----|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7  | 6  | 5   | 4 | 3 | 2 | 1 | 0 |
| 0  | 1  | 0  | 0  | 0  | 1  | 0 | 0 | DN | Rm | Rdn |   |   |   |   |   |

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn); n = d; m = UInt(Rm); setflags = FALSE; (shift_t, shift_n) = (SRTYPE_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```



ARM



CPSR

Thumb

Has Push, Pop

# ARM

```
main: stmfd sp!, {fp, lr}
 add fp, sp, #4
 sub sp, sp, #8
 mov r3, #15
 str r3, [fp, #-8]
 b .L2
.L4: ldr r3, .L5
 ldr r2, [fp, #-8]
 ldr r3, [r3, r2, lsl #2]
 cmp r3, #0
 blt .L3
 ldr r3, [fp, #-12]
 add r3, r3, #1
 str r3, [fp, #-12]
.L3: ldr r3, [fp, #-8]
 sub r3, r3, #1
 str r3, [fp, #-8]
.L2: ldr r3, [fp, #-8]
 cmp r3, #0
 bge .L4
 ldr r0, .L6
 ldr r1, [fp, #-12]
 bl printf
 mov r0, r3
 sub sp, fp, #4
 ldmfd sp!, {fp, lr}
 bx lr
```

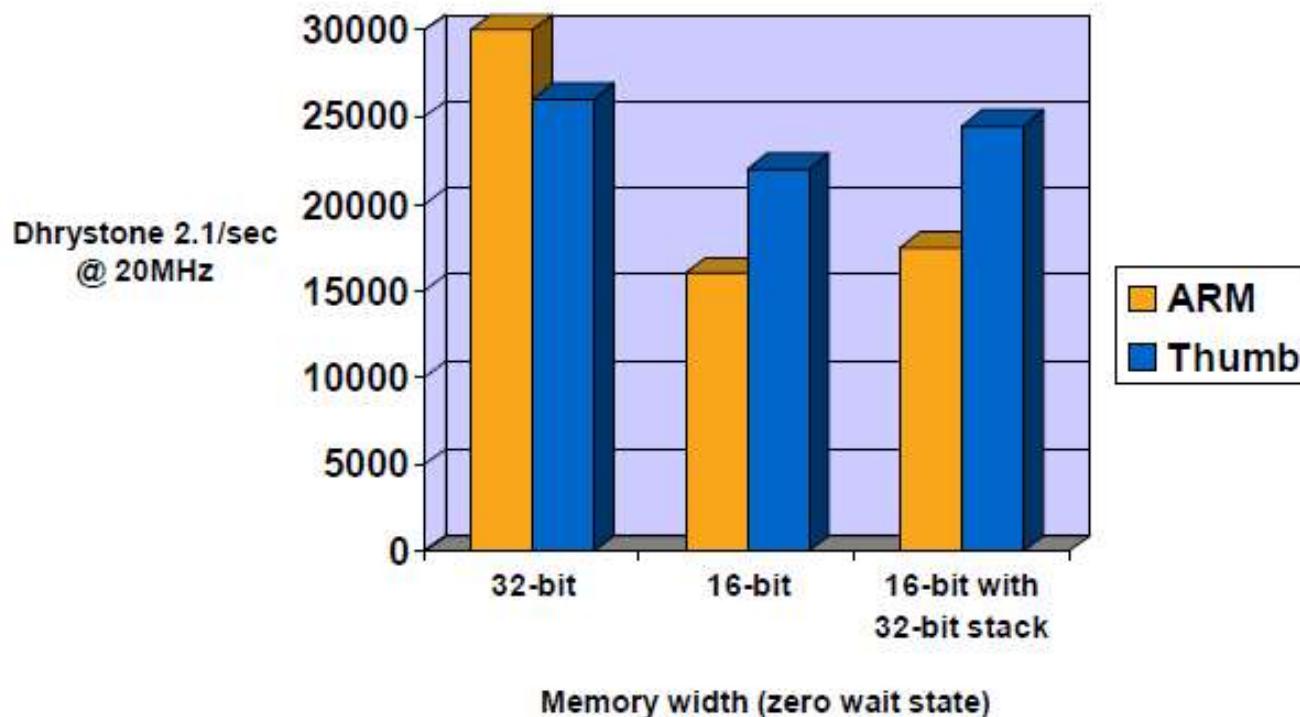
27 instructions x 4B each  
total = 108 Bytes

# Thumb

```
main: push {r7, lr}
 sub sp, sp, #8
 add r7, sp, #0
 mov r3, #15
 str r3, [r7, #4]
 b .L2
.L4: ldr r3, .L5
 ldr r2, [r7, #4]
 lsl r2, r2, #2
 ldr r3, [r2, r3]
 cmp r3, #0
 blt .L3
 ldr r3, [r7]
 add r3, r3, #1
 str r3, [r7]
.L3: ldr r3, [r7, #4]
 sub r3, r3, #1
 str r3, [r7, #4]
.L2: ldr r3, [r7, #4]
 cmp r3, #0
 bge .L4
 ldr r2, .L6
 ldr r3, [r7]
 mov r0, r2
 mov r1, r3
 bl printf
 mov r0, r3
 mov sp, r7
 add sp, sp, #8
 pop {r7}
 pop {r1}
 bx r1
```

32 instructions x 2B each  
total = 64 Bytes

# ARM and Thumb Performance



## Floating Point (!!)

Uses a separate register file

Has either 32 32b registers called S0 to S31  
or 16 64b registers called D0 to D15

To get values from “regular” registers in/out of these, you can use VMOV

**VMOV.size S3, R1**

where **size** is .8 or .16 or .32

**VADD.F32 Sd, Sn, Sm    also VMUL, VDIV, VSUB, VSQRT...**

**VADD.F64 Dd, Dn, Dm**

# **Smart C Programming**

**First 4 parameters passed in registers**

**In loops always make the loop variable count down, if possible**

# Size of local variables

```
int wordsize(int a)
{
 a=a+1;
 return a;
}

int halfsize(short b)
{
 b=b+1;
 return b;
}

int bytesize(char c)
{
 c=c+1;
 return c;
}
```

```
wordsize
ADD r0,r0,#1
BX lr

halfsize
ADD r0,r0,#1
MOV r0,r0,LSL #16
MOV r0,r0,ASR #16
BX lr

bytesize
ADD r0,r0,#1
AND r0,r0,#0xFF
BX lr
```

## Resources

**<http://www.arm.com/support/university/>**

---

# CS 2110 - Lab 02

## Datatypes

Monday, August 30, 2021



---

## Lab Assignment: Canvas Quiz

1. Go to Quizzes on Canvas
2. Select Lab 02, password: **byte**
3. Earn a 100% for attendance credit!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) TAs will give help :)

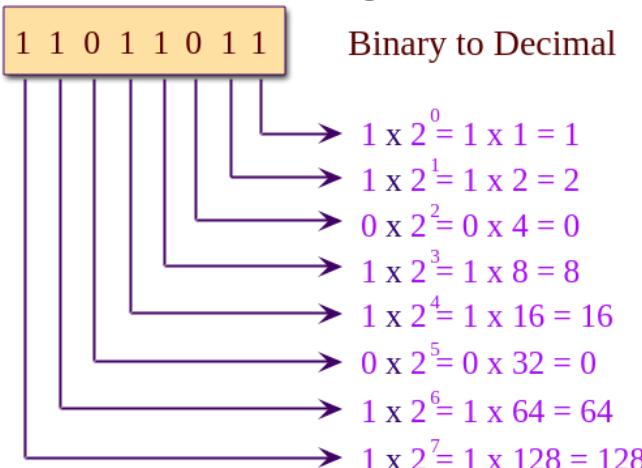


---

# Definitions

- Decimal – our "normal" number counting system (base 10)
  - Represents numbers in terms of powers of 10
  - Example:  $153 = 1(100) + 5(10) + 3(1) = 1(10^2) + 5(10^1) + 3(10^0)$
- Binary – base 2
  - Uses the digits 0 and 1
- Octal – base 8
  - Uses the digits 0-7
- Hexadecimal – base 16
  - Use the digits 0-9 as well as A-F to represent 10-15

# Converting between Decimal and Binary



$$(11011011)_2 = (219)_{10}$$

## Decimal to Binary Conversion

### Divide by 2 Process

$$\text{Decimal } \# 13 \div 2 = 6 \text{ remainder } 1$$

$$6 \div 2 = 3 \text{ remainder } 0$$

Divide-by-2 Process  
Stops When  
Quotient Reaches 0

$$\div 2 = 1 \text{ remainder } 1$$

$$0 \text{ remainder } 1$$

1 1 0 1

---

## Binary Addition

- Just like decimal addition
- $0 + 0 = 0$ , carry out = 0
- $0 + 1 = 1$ , carry out = 0
- $1 + 0 = 1$ , carry out = 0
- $1 + 1 = 0$ , carry out = 1

---

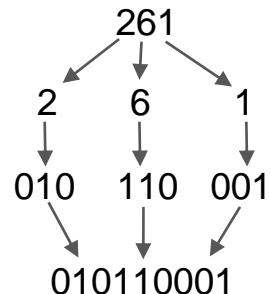
## Practice:

- Let's convert the decimal numbers 149 and 30 to binary
- Use what we learned about binary addition to add them together!
- Once we've done that, convert it back!
- Did we get the answer we expect?

---

# Octal to Binary

1. Split each digit up
2. Convert each digit to 3 bit binary
3. Combine each binary number
4. How can we do the reverse?

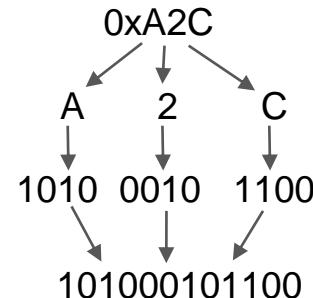


---

# Hexadecimal to Binary

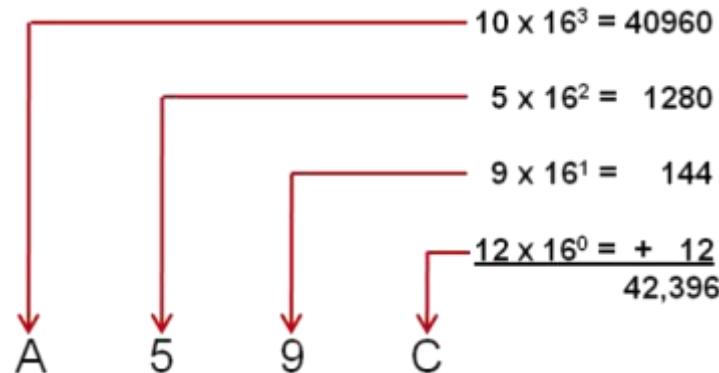
Pretty much the exact same as octal.

1. Split each digit up
2. Convert each digit to 4 bit binary
3. Combine each binary number
4. How can we do the reverse?



## Hexadecimal to Decimal

We can use the same process to convert from octal to decimal as well!



---

# Signed vs Unsigned

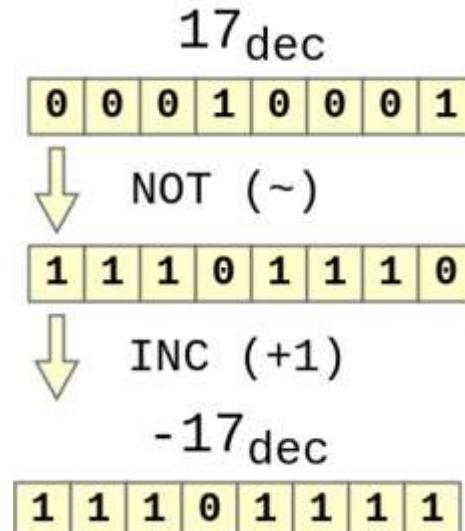
Can unsigned binary represent negative numbers?

How do we represent negative numbers in binary?

- Signed magnitude
- 1's complement
- 2's complement

# 2's Complement

| Bits | Unsigned Value | Two's Complement Value |
|------|----------------|------------------------|
| 000  | 0              | 0                      |
| 001  | 1              | 1                      |
| 010  | 2              | 2                      |
| 011  | 3              | 3                      |
| 100  | 4              | -4                     |
| 101  | 5              | -3                     |
| 110  | 6              | -2                     |
| 111  | 7              | -1                     |



Two's complement  
negation:

$$-n == (\sim n) + 1$$



# Detecting Overflow

**Overflow:** what if I don't have enough bits to store the output of my addition?

For example,  $1111 + 0001 = 10000$  – but what if I only have four bits to represent a number?

**Unsigned addition** – if there is a carry-out from the addition, then overflow has occurred

Example:  $6 + 12 = 0110 + 1100 = (1)0010 = 2$  (carry out dropped)

**Signed addition** – if you get the wrong sign, then overflow has occurred.

There are two methods of detecting this:

1. Adding two numbers with the same sign resulting in the opposite sign. Adding two numbers with opposite sign never overflows—why?
2. If the carry in and the carry out of the last bit are **different**

Example:  $6 + 3 = 0110 + 0011 = 1001 = -7$

---

## Logical Operators

- NOT (Symbol : ~)
- AND (Symbol : &)
- OR (Symbol : |)
- XOR (Symbol : ^)
- A NOR B = NOT (A OR B) = ~(A | B)
- A NAND B = NOT (A AND B) = ~(A & B)



# Masking

Using the operators we just learned, how do we isolate or remove a specific part of a binary integer?

- To **SET** a bit means to make it a **1**
- To **CLEAR** a bit means to make it a **0**

---

## Masking

Using the operators (OR, NOT, XOR, AND) we just went over, how can we...

- Extract the specified bits (make the rest all 0)
- Clear the specified bits (make *them* all 0)
- Set the specified
- Flip the specified bits

1101 0110 0100 1001

---

## Shifting

- Left shifting (`<<`)
- Right shifting (signed and unsigned) (`>>`, `>>>`)

These operations are equivalent to multiplying or dividing by two;  
why?

How could we use this to get only the highlighted part of the  
integer below?

A = 0000 0000 0000 **0110**

# ASCII

- ASCII : American standard code for information interchange
- Each character on your computer (a, z, Z, 1, \$ etc.) are associated with a unique binary integer dictated by this code

| Dec | Hex | Name              | Char | Ctrl-char | Dec | Hex | Char  | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|-------------------|------|-----------|-----|-----|-------|-----|-----|------|-----|-----|------|
| 0   | 0   | Null              | NUL  | CTRL-@    | 32  | 20  | Space | 64  | 40  | @    | 96  | 60  | '    |
| 1   | 1   | Start of heading  | SOH  | CTRL-A    | 33  | 21  | !     | 65  | 41  | A    | 97  | 61  | a    |
| 2   | 2   | Start of text     | STX  | CTRL-B    | 34  | 22  | "     | 66  | 42  | B    | 98  | 62  | b    |
| 3   | 3   | End of text       | ETX  | CTRL-C    | 35  | 23  | #     | 67  | 43  | C    | 99  | 63  | c    |
| 4   | 4   | End of xmit       | EOT  | CTRL-D    | 36  | 24  | \$    | 68  | 44  | D    | 100 | 64  | d    |
| 5   | 5   | Enquiry           | ENQ  | CTRL-E    | 37  | 25  | %     | 69  | 45  | E    | 101 | 65  | e    |
| 6   | 6   | Acknowledge       | ACK  | CTRL-F    | 38  | 26  | &     | 70  | 46  | F    | 102 | 66  | f    |
| 7   | 7   | Bell              | BEL  | CTRL-G    | 39  | 27  | '     | 71  | 47  | G    | 103 | 67  | g    |
| 8   | 8   | Backspace         | BS   | CTRL-H    | 40  | 28  | (     | 72  | 48  | H    | 104 | 68  | h    |
| 9   | 9   | Horizontal tab    | HT   | CTRL-I    | 41  | 29  | )     | 73  | 49  | I    | 105 | 69  | i    |
| 10  | 0A  | Line feed         | LF   | CTRL-J    | 42  | 2A  | *     | 74  | 4A  | J    | 106 | 6A  | j    |
| 11  | 0B  | Vertical tab      | VT   | CTRL-K    | 43  | 2B  | +     | 75  | 4B  | K    | 107 | 6B  | k    |
| 12  | 0C  | Form feed         | FF   | CTRL-L    | 44  | 2C  | ,     | 76  | 4C  | L    | 108 | 6C  | l    |
| 13  | 0D  | Carriage feed     | CR   | CTRL-M    | 45  | 2D  | -     | 77  | 4D  | M    | 109 | 6D  | m    |
| 14  | 0E  | Shift out         | SO   | CTRL-N    | 46  | 2E  | .     | 78  | 4E  | N    | 110 | 6E  | n    |
| 15  | 0F  | Shift in          | SI   | CTRL-O    | 47  | 2F  | /     | 79  | 4F  | O    | 111 | 6F  | o    |
| 16  | 10  | Data line escape  | DLE  | CTRL-P    | 48  | 30  | 0     | 80  | 50  | P    | 112 | 70  | p    |
| 17  | 11  | Device control 1  | DC1  | CTRL-Q    | 49  | 31  | 1     | 81  | 51  | Q    | 113 | 71  | q    |
| 18  | 12  | Device control 2  | DC2  | CTRL-R    | 50  | 32  | 2     | 82  | 52  | R    | 114 | 72  | r    |
| 19  | 13  | Device control 3  | DC3  | CTRL-S    | 51  | 33  | 3     | 83  | 53  | S    | 115 | 73  | s    |
| 20  | 14  | Device control 4  | DC4  | CTRL-T    | 52  | 34  | 4     | 84  | 54  | T    | 116 | 74  | t    |
| 21  | 15  | Neg acknowledge   | NAK  | CTRL-U    | 53  | 35  | 5     | 85  | 55  | U    | 117 | 75  | u    |
| 22  | 16  | Synchronous idle  | SYN  | CTRL-V    | 54  | 36  | 6     | 86  | 56  | V    | 118 | 76  | v    |
| 23  | 17  | End of xmit block | ETB  | CTRL-W    | 55  | 37  | 7     | 87  | 57  | W    | 119 | 77  | w    |
| 24  | 18  | Cancel            | CAN  | CTRL-X    | 56  | 38  | 8     | 88  | 58  | X    | 120 | 78  | x    |
| 25  | 19  | End of medium     | EM   | CTRL-Y    | 57  | 39  | 9     | 89  | 59  | Y    | 121 | 79  | y    |
| 26  | 1A  | Substitute        | SUB  | CTRL-Z    | 58  | 3A  | :     | 90  | 5A  | Z    | 122 | 7A  | z    |
| 27  | 1B  | Escape            | ESC  | CTRL-[    | 59  | 3B  | :     | 91  | 5B  | [    | 123 | 7B  | {    |
| 28  | 1C  | File separator    | FS   | CTRL-\    | 60  | 3C  | <     | 92  | 5C  | \    | 124 | 7C  |      |
| 29  | 1D  | Group separator   | GS   | CTRL-]    | 61  | 3D  | =     | 93  | 5D  | ]    | 125 | 7D  | }    |
| 30  | 1E  | Record separator  | RS   | CTRL-^    | 62  | 3E  | >     | 94  | 5E  | ^    | 126 | 7E  | ~    |
| 31  | 1F  | Unit separator    | US   | CTRL-_    | 63  | 3F  | ?     | 95  | 5F  | _    | 127 | 7F  | DEL  |



# ASCII

Example:

'a' is equivalent to 97, or 0110 0001

'A' is equivalent to 65, or 0100 0001

Notice that these differ by exactly 32... How can we use masking to convert from uppercase to lowercase?

---

# CS 2110 - Lab 03

IEEE 754 & Digital Logic

Wednesday, September 1, 2021



---

# Homework 1

- Releasing tomorrow!
- Due Wednesday, September 8<sup>rd</sup> at 11:59 PM
  - Standard 24-hour grace period as described in the syllabus
  - Submit by Thursday, September 9th at 11:59 PM for 25% penalty
- Files available on Canvas
- Submit on Gradescope
  - Double check that your grade on Gradescope is your desired grade!

---

## Lab Assignment: Canvas Quiz

1. Go to Quizzes on Canvas
2. Select Lab 03, password: **IEEE-754**
3. Earn 100% for attendance credit!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



# ASCII Review

- American Standard Code for Information Interchange
- Each character on your computer (a, z, Z, 1, \$ etc.) are associated with a unique binary integer dictated by this code

| Dec | Hex | Name              | Char | Ctrl-char | Dec | Hex | Char  | Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|-------------------|------|-----------|-----|-----|-------|-----|-----|------|-----|-----|------|
| 0   | 0   | Null              | NUL  | CTRL-@    | 32  | 20  | Space | 64  | 40  | @    | 96  | 60  | '    |
| 1   | 1   | Start of heading  | SOH  | CTRL-A    | 33  | 21  | !     | 65  | 41  | A    | 97  | 61  | a    |
| 2   | 2   | Start of text     | STX  | CTRL-B    | 34  | 22  | "     | 66  | 42  | B    | 98  | 62  | b    |
| 3   | 3   | End of text       | ETX  | CTRL-C    | 35  | 23  | #     | 67  | 43  | C    | 99  | 63  | c    |
| 4   | 4   | End of xmit       | EOT  | CTRL-D    | 36  | 24  | \$    | 68  | 44  | D    | 100 | 64  | d    |
| 5   | 5   | Enquiry           | ENQ  | CTRL-E    | 37  | 25  | %     | 69  | 45  | E    | 101 | 65  | e    |
| 6   | 6   | Acknowledge       | ACK  | CTRL-F    | 38  | 26  | &     | 70  | 46  | F    | 102 | 66  | f    |
| 7   | 7   | Bell              | BEL  | CTRL-G    | 39  | 27  | '     | 71  | 47  | G    | 103 | 67  | g    |
| 8   | 8   | Backspace         | BS   | CTRL-H    | 40  | 28  | (     | 72  | 48  | H    | 104 | 68  | h    |
| 9   | 9   | Horizontal tab    | HT   | CTRL-I    | 41  | 29  | )     | 73  | 49  | I    | 105 | 69  | i    |
| 10  | 0A  | Line feed         | LF   | CTRL-J    | 42  | 2A  | *     | 74  | 4A  | J    | 106 | 6A  | j    |
| 11  | 0B  | Vertical tab      | VT   | CTRL-K    | 43  | 2B  | +     | 75  | 4B  | K    | 107 | 6B  | k    |
| 12  | 0C  | Form feed         | FF   | CTRL-L    | 44  | 2C  | ,     | 76  | 4C  | L    | 108 | 6C  | l    |
| 13  | 0D  | Carriage feed     | CR   | CTRL-M    | 45  | 2D  | -     | 77  | 4D  | M    | 109 | 6D  | m    |
| 14  | 0E  | Shift out         | SO   | CTRL-N    | 46  | 2E  | .     | 78  | 4E  | N    | 110 | 6E  | n    |
| 15  | 0F  | Shiftin           | SI   | CTRL-O    | 47  | 2F  | /     | 79  | 4F  | O    | 111 | 6F  | o    |
| 16  | 10  | Data line escape  | DLE  | CTRL-P    | 48  | 30  | 0     | 80  | 50  | P    | 112 | 70  | p    |
| 17  | 11  | Device control 1  | DC1  | CTRL-Q    | 49  | 31  | 1     | 81  | 51  | Q    | 113 | 71  | q    |
| 18  | 12  | Device control 2  | DC2  | CTRL-R    | 50  | 32  | 2     | 82  | 52  | R    | 114 | 72  | r    |
| 19  | 13  | Device control 3  | DC3  | CTRL-S    | 51  | 33  | 3     | 83  | 53  | S    | 115 | 73  | s    |
| 20  | 14  | Device control 4  | DC4  | CTRL-T    | 52  | 34  | 4     | 84  | 54  | T    | 116 | 74  | t    |
| 21  | 15  | Neg acknowledge   | NAK  | CTRL-U    | 53  | 35  | 5     | 85  | 55  | U    | 117 | 75  | u    |
| 22  | 16  | Synchronous idle  | SYN  | CTRL-V    | 54  | 36  | 6     | 86  | 56  | V    | 118 | 76  | v    |
| 23  | 17  | End of xmit block | ETB  | CTRL-W    | 55  | 37  | 7     | 87  | 57  | W    | 119 | 77  | w    |
| 24  | 18  | Cancel            | CAN  | CTRL-X    | 56  | 38  | 8     | 88  | 58  | X    | 120 | 78  | x    |
| 25  | 19  | End of medium     | EM   | CTRL-Y    | 57  | 39  | 9     | 89  | 59  | Y    | 121 | 79  | y    |
| 26  | 1A  | Substitute        | SUB  | CTRL-Z    | 58  | 3A  | :     | 90  | 5A  | Z    | 122 | 7A  | z    |
| 27  | 1B  | Escape            | ESC  | CTRL-[    | 59  | 3B  | :     | 91  | 5B  | [    | 123 | 7B  | {    |
| 28  | 1C  | File separator    | FS   | CTRL-\    | 60  | 3C  | <     | 92  | 5C  | \    | 124 | 7C  |      |
| 29  | 1D  | Group separator   | GS   | CTRL-]    | 61  | 3D  | =     | 93  | 5D  | ]    | 125 | 7D  | }    |
| 30  | 1E  | Record separator  | RS   | CTRL-^    | 62  | 3E  | >     | 94  | 5E  | ^    | 126 | 7E  | ~    |
| 31  | 1F  | Unit separator    | US   | CTRL-_    | 63  | 3F  | ?     | 95  | 5F  | _    | 127 | 7F  | DEL  |

| Dec | Hex | Char  |
|-----|-----|-------|
| 32  | 20  | Space |
| 33  | 21  | !     |
| 34  | 22  | "     |
| 35  | 23  | #     |
| 36  | 24  | \$    |
| 37  | 25  | %     |
| 38  | 26  | &     |
| 39  | 27  | *     |
| 40  | 28  | (     |
| 41  | 29  | )     |
| 42  | 2A  | *     |
| 43  | 2B  | +     |
| 44  | 2C  | :     |
| 45  | 2D  | -     |
| 46  | 2E  | ,     |
| 47  | 2F  | /     |
| 48  | 30  | 0     |
| 49  | 31  | 1     |
| 50  | 32  | 2     |
| 51  | 33  | 3     |
| 52  | 34  | 4     |
| 53  | 35  | 5     |
| 54  | 36  | 6     |
| 55  | 37  | 7     |
| 56  | 38  | 8     |
| 57  | 39  | 9     |
| 58  | 3A  | :     |
| 59  | 3B  | =     |
| 60  | 3C  | <     |
| 61  | 3D  | #     |
| 62  | 3E  | >     |
| 63  | 3F  | ?     |

## ASCII Tips and Tricks

- Converting from numeric characters to numbers
  - ASCII numbers are ordered increasingly, starting with 48 == '0'
  - We can take advantage of this fact to convert from ASCII codes to numbers easily
- What is the value of `i` below? Is this correct? If not, what should we do instead?
  - `char c = '7'; // any '0' thru '9'`
  - `int i = (int) c;`

---

# ASCII Tips and Tricks

- Converting between upper/lower case
  - Corresponding letters differ by exactly 32
  - To convert from upper to lower case, add 32
  - To convert from lower to upper case, subtract 32
  - Alternatively:
    - Note that all uppercase letters are <96 and lowercase are >96.
    - We can simply toggle bit 5, as it is always unset for capital letters and set for lowercase ones.

| Dec | Hex | Char | Dec | Hex | Char |
|-----|-----|------|-----|-----|------|
| 64  | 40  | @    | 96  | 60  | '    |
| 65  | 41  | A    | 97  | 61  | a    |
| 66  | 42  | B    | 98  | 62  | b    |
| 67  | 43  | C    | 99  | 63  | c    |
| 68  | 44  | D    | 100 | 64  | d    |
| 69  | 45  | E    | 101 | 65  | e    |
| 70  | 46  | F    | 102 | 66  | f    |
| 71  | 47  | G    | 103 | 67  | g    |
| 72  | 48  | H    | 104 | 68  | h    |
| 73  | 49  | I    | 105 | 69  | i    |
| 74  | 4A  | J    | 106 | 6A  | j    |
| 75  | 4B  | K    | 107 | 6B  | k    |
| 76  | 4C  | L    | 108 | 6C  | l    |
| 77  | 4D  | M    | 109 | 6D  | m    |
| 78  | 4E  | N    | 110 | 6E  | n    |
| 79  | 4F  | O    | 111 | 6F  | o    |
| 80  | 50  | P    | 112 | 70  | p    |
| 81  | 51  | Q    | 113 | 71  | q    |
| 82  | 52  | R    | 114 | 72  | r    |
| 83  | 53  | S    | 115 | 73  | s    |
| 84  | 54  | T    | 116 | 74  | t    |
| 85  | 55  | U    | 117 | 75  | u    |
| 86  | 56  | V    | 118 | 76  | v    |
| 87  | 57  | W    | 119 | 77  | w    |
| 88  | 58  | X    | 120 | 78  | x    |
| 89  | 59  | Y    | 121 | 79  | y    |
| 90  | 5A  | Z    | 122 | 7A  | z    |
| 91  | 5B  | [    | 123 | 7B  | {    |
| 92  | 5C  | \    | 124 | 7C  |      |
| 93  | 5D  | ]    | 125 | 7D  | }    |
| 94  | 5E  | ^    | 126 | 7E  | ~    |
| 95  | 5F  | _    | 127 | 7F  | DEL  |

# IEEE 754 Floating Point Numbers

- What is the sign?      Formula:  $(-1)^S \times 1.M \times 2^{E-127}$
  - What is the exponent?
  - What is the mantissa?
  - What is the range of each?  
(hint: consider which is signed)
- 
- The diagram illustrates the IEEE 754 Single Precision floating-point format. It consists of three adjacent boxes: 'Sign' (1 bit), 'Exponent' (8 bits), and 'Mantissa' (23 bits). A horizontal arrow above the boxes spans the entire width and is labeled '32 Bits'. Below each box is a double-headed arrow indicating its bit width: '1 Bit' for the Sign, '8 Bits' for the Exponent, and '23 Bits' for the Mantissa.
- Single Precision  
IEEE 754 Floating-Point Standard

---

## Practice!

Formula:  $(-1)^S * 1.M * 2^{(E-127)}$

1100000001100000000000000000000000000000

Sign  
Exponent  
Mantissa

# Comparing IEEE-754 Numbers

- Treat like a signed, 32 bit whole number (go bitwise)
  - This is why the signed exponent is not defined using 2s complement—it allows us to compare bitwise, rather than having to do a 2's complement comparison on the exponent portion

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |   |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |



# IEEE 754 Edge Cases

Formula:  $(-1)^S \times 1.M \times 2^{(E-127)}$

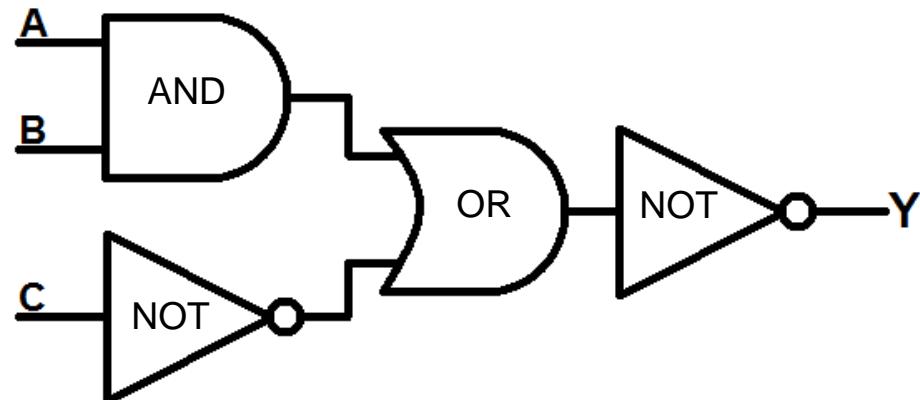
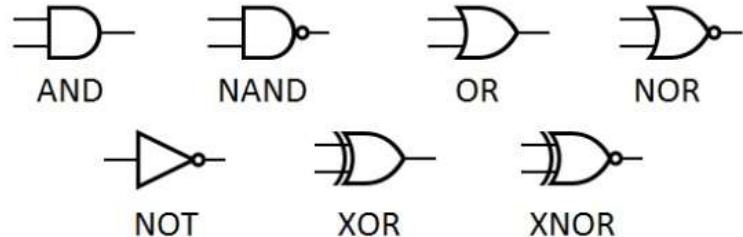
|      | E==0           | 0<E<255         | E==255   |
|------|----------------|-----------------|----------|
| M==0 | 0              | Powers of 2     | infinity |
| M!=0 | Non-normalized | Regular numbers | NAN      |

Non-normalized formula:  $(-1)^S \times 0.M \times 2^{-126}$

note:  $0.M$  and  $2^{-126}$  instead of  $1.M$  and  $2^{-127}$

# Logic Gates

- Abstraction for representing groups of transistors
- Perform basic binary operations
- Basic building blocks of circuits
  - Circuits contain at least one input and at least one output
  - Wires and logic gates connect the outputs to the input
  - Ex: what is the value of Y if A,B,C are all 1?



---

## Converting a Truth Table to Sum of Products

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | 1           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 1           |
| 0 | 1 | 1 | 0           |
| 1 | 0 | 0 | 1           |
| 1 | 0 | 1 | 0           |
| 1 | 1 | 0 | 1           |
| 1 | 1 | 1 | 1           |

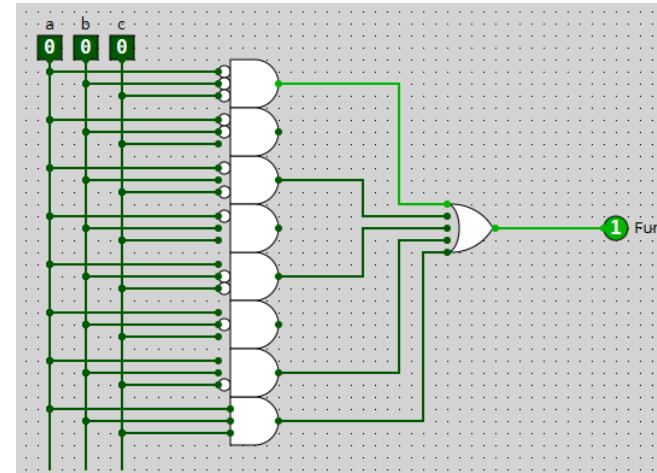
- Create a term for each case that results in 1
  - Each term is an AND
  - e.g. A=0, B=1, C=0 becomes  $A'BC'$
- OR all the terms together

---

## Sum of Products Circuit From a Truth Table

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | 1           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 1           |
| 0 | 1 | 1 | 0           |
| 1 | 0 | 0 | 1           |
| 1 | 0 | 1 | 0           |
| 1 | 1 | 0 | 1           |
| 1 | 1 | 1 | 1           |

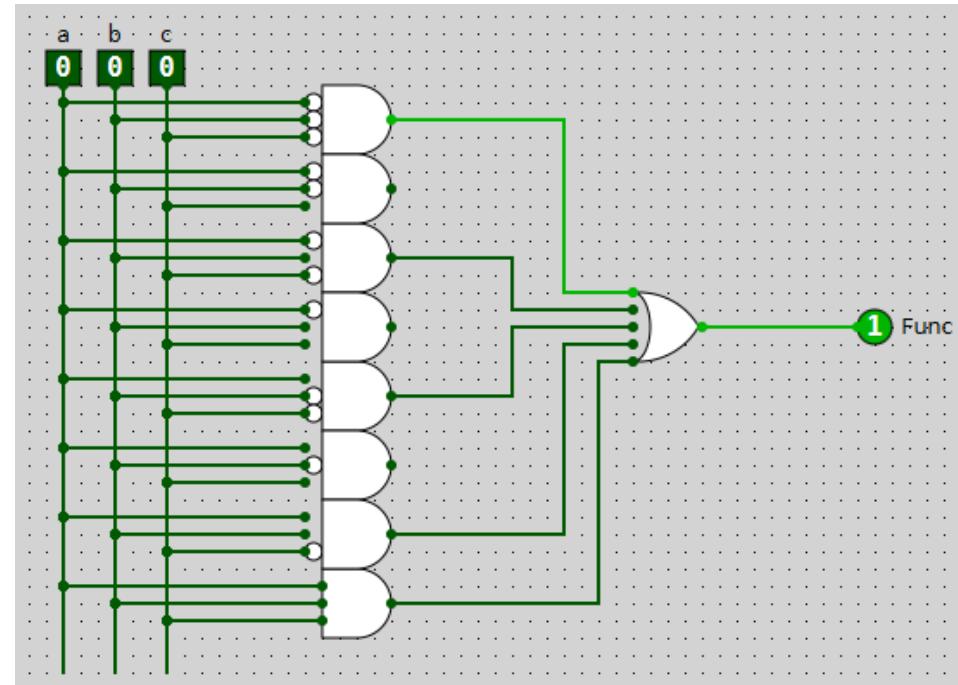
We can build out a circuit representation of the previous sum-of-products expression using many AND gates and connecting them together with an OR



---

## Sum of Products Circuit From a Truth Table

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | 1           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 1           |
| 0 | 1 | 1 | 0           |
| 1 | 0 | 0 | 1           |
| 1 | 0 | 1 | 0           |
| 1 | 1 | 0 | 1           |
| 1 | 1 | 1 | 1           |



Example:  $A \mid (B \ \& \ \sim C)$

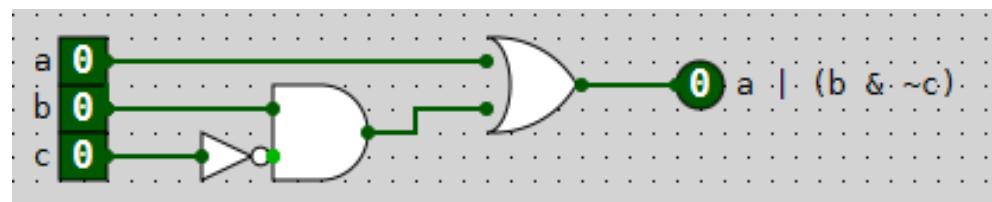
---

## Converting a Boolean Expression

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | 0           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 1           |
| 0 | 1 | 1 | 0           |
| 1 | 0 | 0 | 1           |
| 1 | 0 | 1 | 1           |
| 1 | 1 | 0 | 1           |
| 1 | 1 | 1 | 1           |

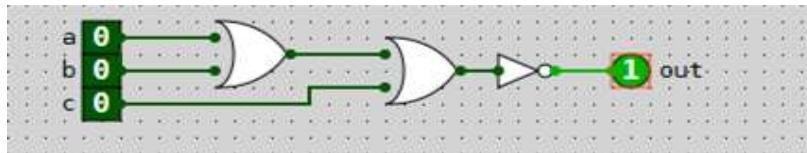
To build the truth table, fill out the first 3 columns of the table and plug the inputs into the expression.

To build the circuit, you can draw each of the gates from the expression and connect the pins



---

## Converting from a Circuit



From a circuit, we can easily get the expression by looking at the gates:

$$\sim((A \mid B) \mid C)$$

You can then build the truth table using the expression, or by just plugging in inputs into the circuit

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | 1           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 0           |
| 0 | 1 | 1 | 0           |
| 1 | 0 | 0 | 0           |
| 1 | 0 | 1 | 0           |
| 1 | 1 | 0 | 0           |
| 1 | 1 | 1 | 0           |

---

# CS 2110 - Lab 04

## Transistors, Gates, & K-Maps

Wednesday, September 8, 2021



---

## Lab Assignment: CircuitSim Tutorial

- 1) Complete Lab Summary quiz on Canvas
  - o Password: CMOS
- 2) Files on Canvas under: Lab Assignments -> Lab04
  - a) Extract the folder to the same directory as your `cs2110docker.sh` file
  - b) Boot up Docker and launch the CircuitSim icon
- 3) After the slides are done, complete the lab assignment  
(show a TA and you're good to go!)

---

# Homework 1

- Released last Thursday!
- **Due tonight at 11:59 PM**
  - Standard grace period until tomorrow at 11:59 PM with 25% deduction applies
- Files available on Canvas
- Submit on Gradescope
  - **Double check that your grade on Gradescope is your desired grade!**

---

## Homework 2

- Released Thursday, September 9<sup>th</sup> (tomorrow)!
- **Due Wednesday, September 15<sup>th</sup> at 11:59 PM**
  - Standard grace period with 25% deduction until 9/16 applies
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)

---

## Quiz 1

- Next Monday (September 13<sup>th</sup>)
- You must come to class and take the quiz here
- Open book, open note, open internet
  - Use one device only
  - No communication with anyone else except your TAs
- Full 75 minutes to take the quiz!
- A topic list will be posted on Canvas

---

## Academic Honesty Policy

- We take any infringements on the academic honesty policy very seriously; if found cheating, you will likely get a 0 and be reported to the Office of Student Integrity
- Taking the test anywhere besides your lab section or sending anyone the quiz code, etc. is considered an academic honesty violation

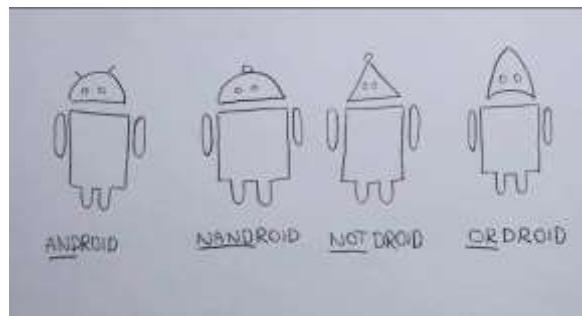
---

## Today's Topics

- Transistors and CMOS Design
- DeMorgan's Law
- Decoders and Multiplexers
- Karnaugh Maps (K-Maps)

# Logical Operations

How can we implement these different operations in hardware?

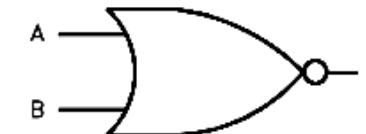


AND Gate



| INPUT |   | OUTPUT |
|-------|---|--------|
| A     | B | F      |
| 0     | 0 | 0      |
| 0     | 1 | 0      |
| 1     | 0 | 0      |
| 1     | 1 | 1      |

NOR Gate



| INPUT |   | OUTPUT |
|-------|---|--------|
| A     | B | F      |
| 0     | 0 | 1      |
| 0     | 1 | 0      |
| 1     | 0 | 0      |
| 1     | 1 | 0      |

OR Gate



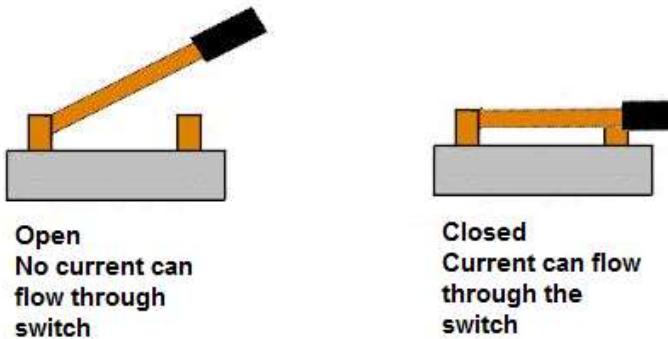
| INPUT |   | OUTPUT |
|-------|---|--------|
| A     | B | F      |
| 0     | 0 | 0      |
| 0     | 1 | 1      |
| 1     | 0 | 1      |
| 1     | 1 | 1      |

Exclusive OR Gate



| INPUT |   | OUTPUT |
|-------|---|--------|
| A     | B | C      |
| 0     | 0 | 0      |
| 0     | 1 | 1      |
| 1     | 0 | 1      |
| 1     | 1 | 0      |

# Transistors



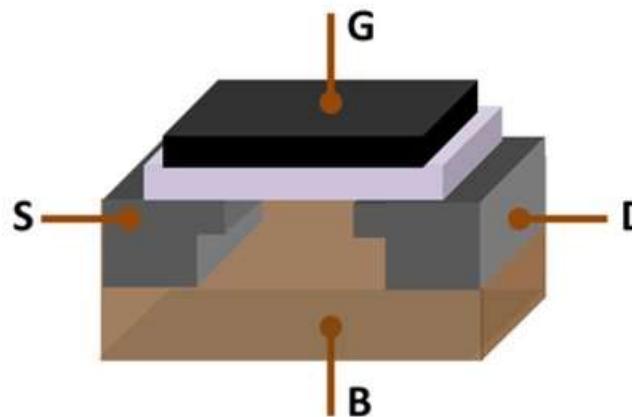
Transistors are digital “switches” and are the magical building blocks of all gates.

Parts of a MOS transistor:

- Gate
- Source
- Drain

Types of MOS transistors:

- P-type
- N-type

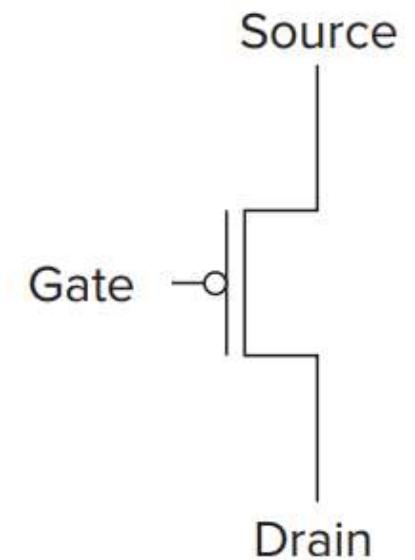
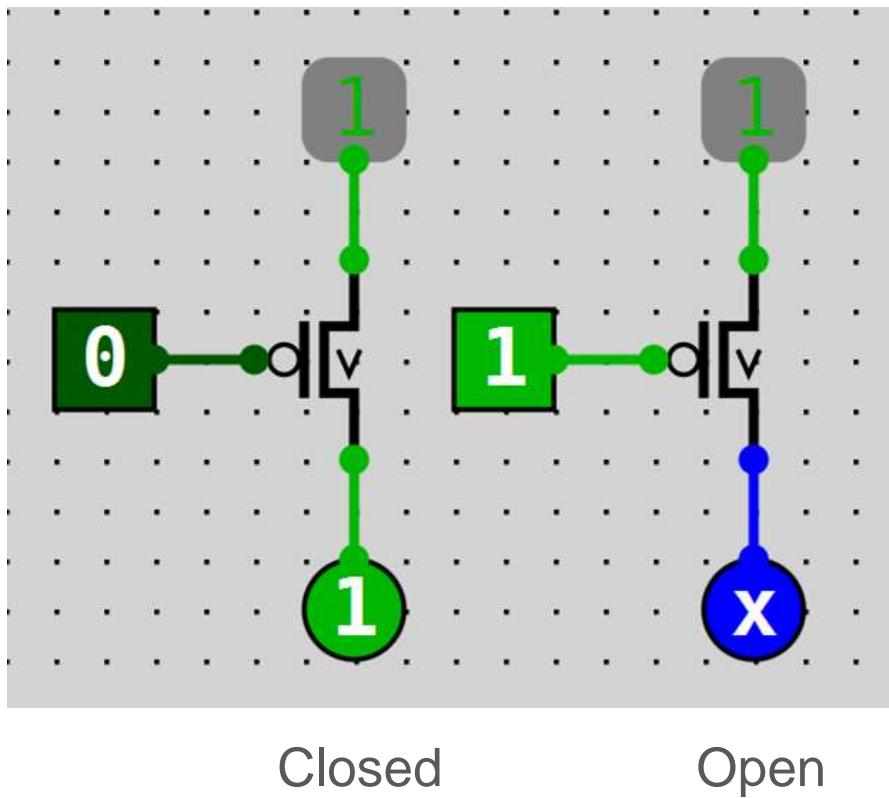


B = Body (irrelevant  
chemical magic!)

Complementary MOS (“CMOS”) design: combining PMOS and NMOS transistors to make more complicated circuits

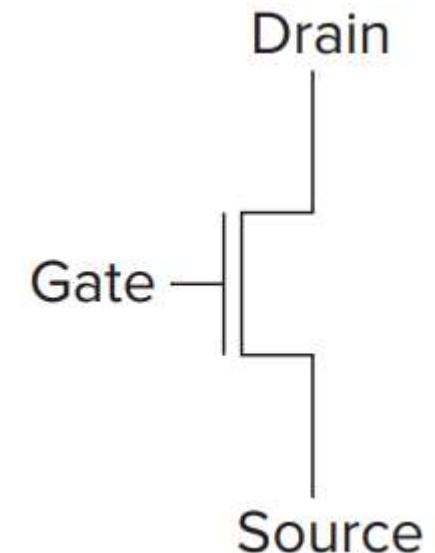
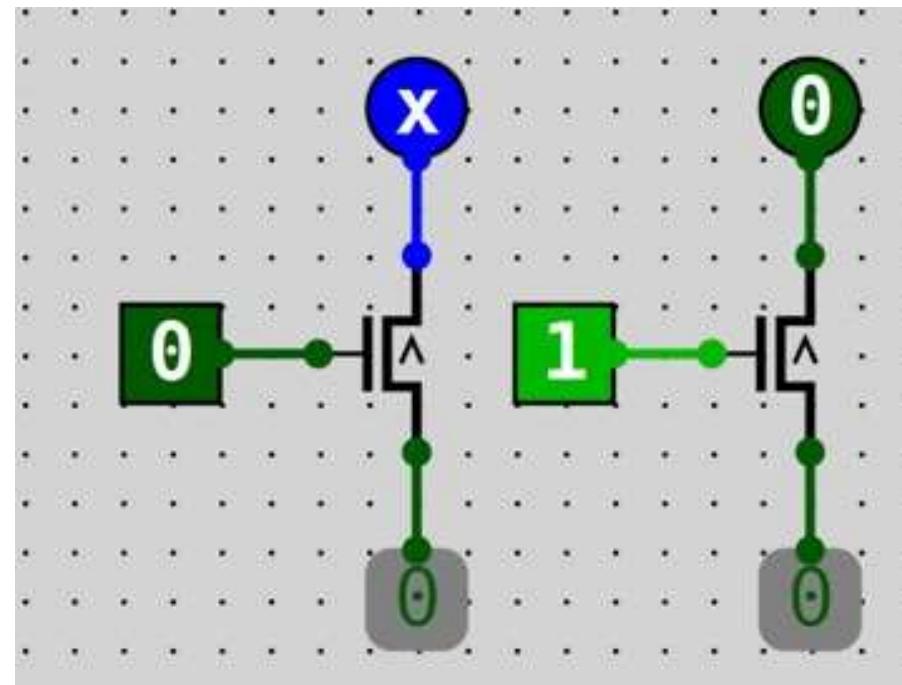
# P-type Transistors

- Connected to Power
  - P-type transistors CANNOT propagate a strong “0” signal
- “Normally closed”



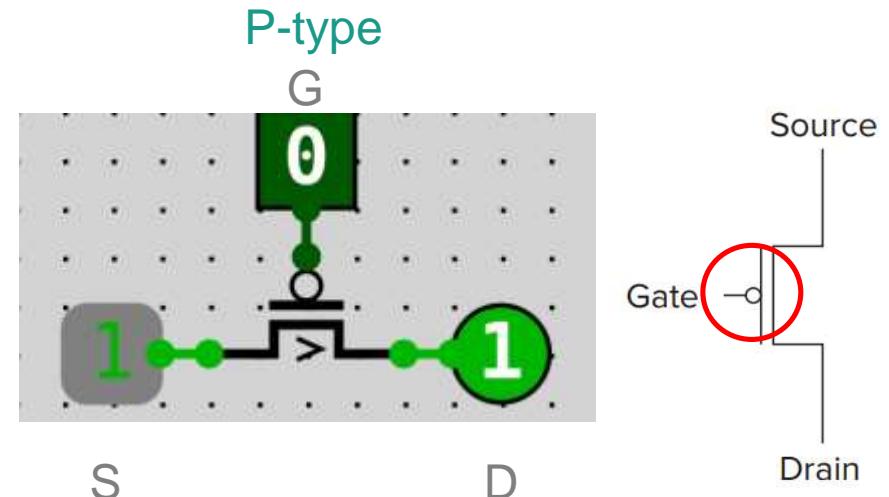
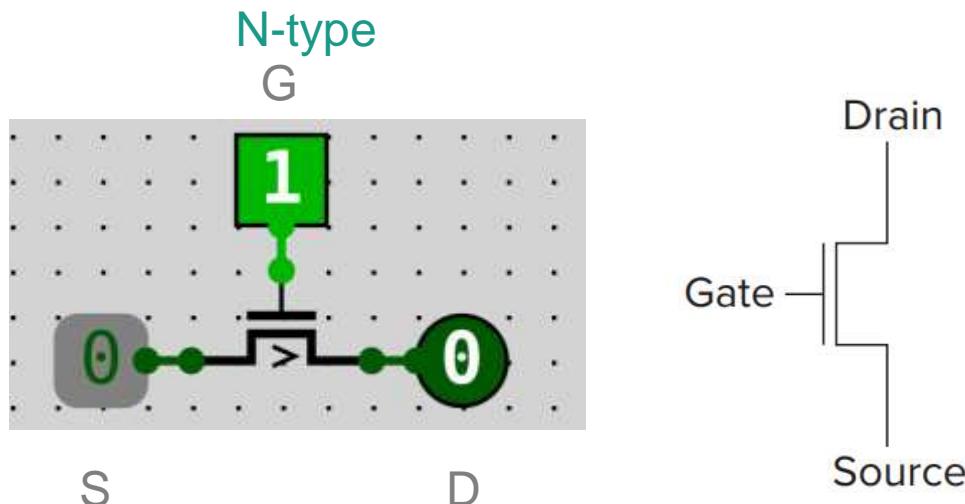
# N-type Transistors

- Connected to Ground
  - N-type transistors CANNOT propagate a strong “1” signal
- “Normally open”



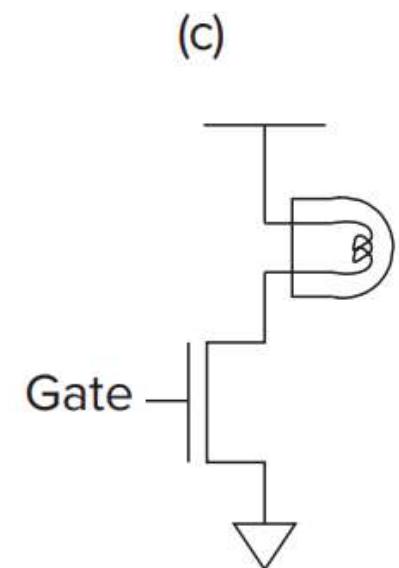
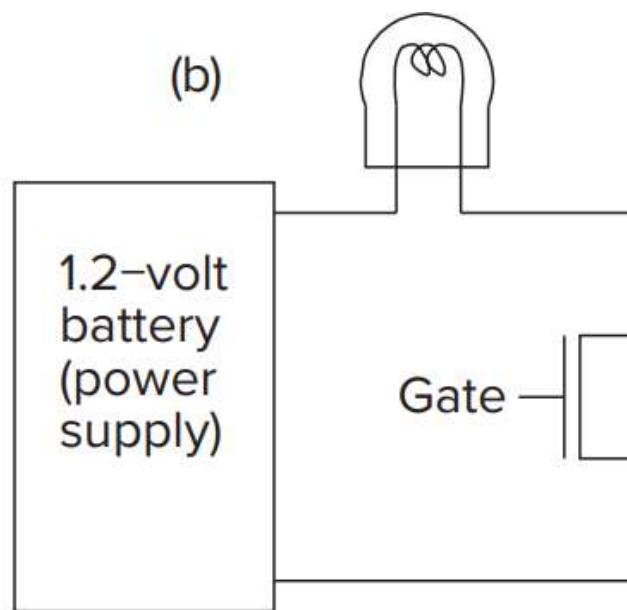
# N-Type vs P-Type

- They have the opposite effect; notice the bubble!
- Check the direction of the arrows – we always point from source (input) to drain (output)
- In CMOS design, connecting multiple P-type in series implies we will connect some *complementary* N-type transistors in parallel, and vice versa



# Transistors

- What transistor type is in this circuit?
- Figure (c) is shorthand for (b)



---

## DeMorgan's Law

(not A and not B) = not (A or B)

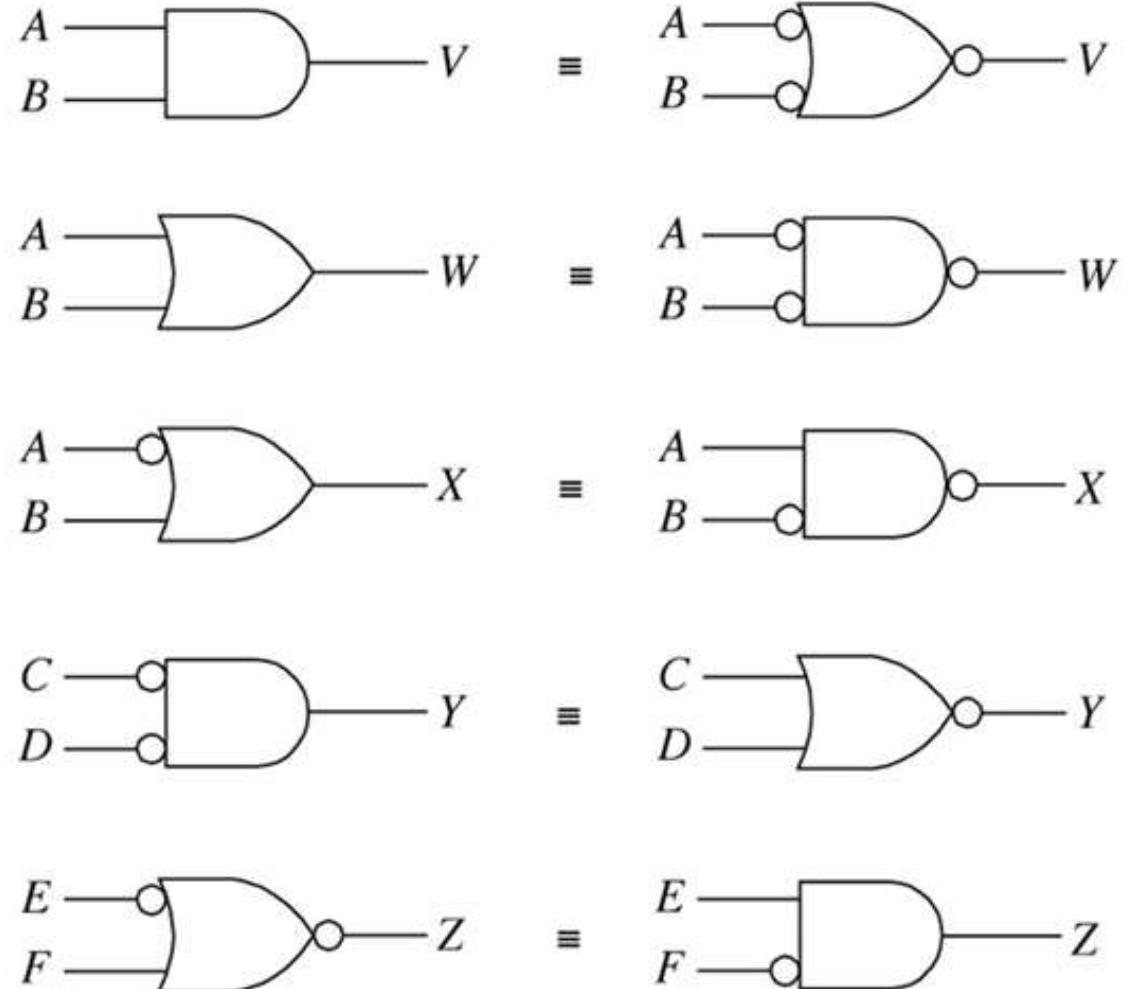
$$\overline{A} \cdot \overline{B} = \overline{A + B}$$

(not A or not B) = not (A and B)

$$\overline{A} + \overline{B} = \overline{A \cdot B}$$

## Conte Bubble Theorem

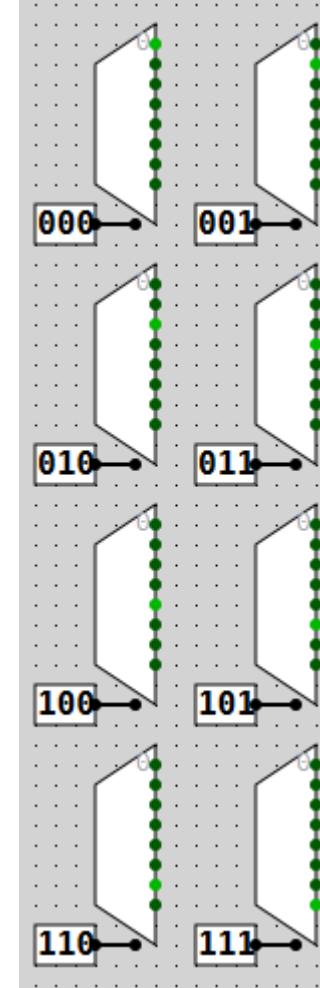
- Switch AND & OR
- Bubble  $\rightarrow$  No Bubble
- No Bubble  $\rightarrow$  Bubble
- They are equivalent!



---

## Decoders

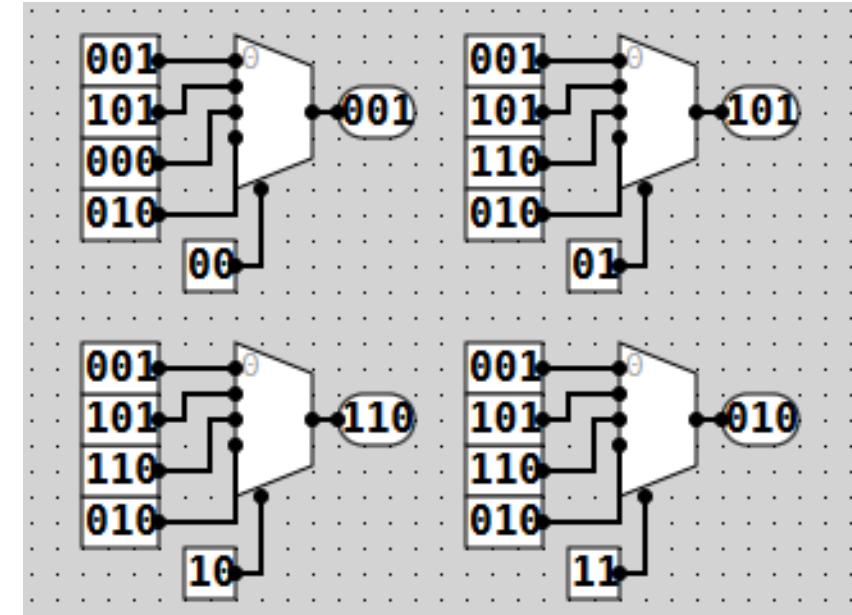
- Sets exactly one output based on which of the input bits are set
- If there are  $n$  input bits then there are  $2^n$  outputs. (Why?)



---

# Multiplexer (mux)

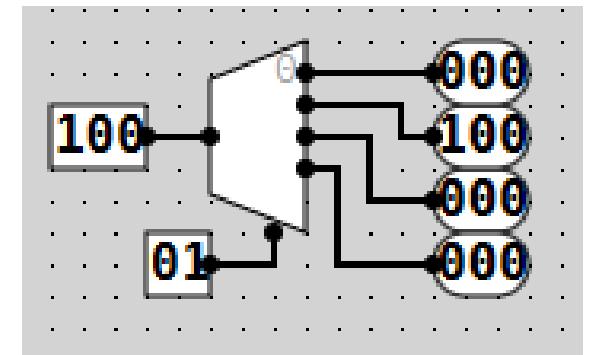
- Selects between inputs using a selector
- If there are  $2^n$  inputs, then there are  $n$  selector bits
- There is always just one output



---

## Demultiplexer (demux)

- Sends the input across exactly one of the output lines
- Other outputs remain zero
- If there are  $2^n$  outputs, then there are  $n$  selector bits
- There is always just one input

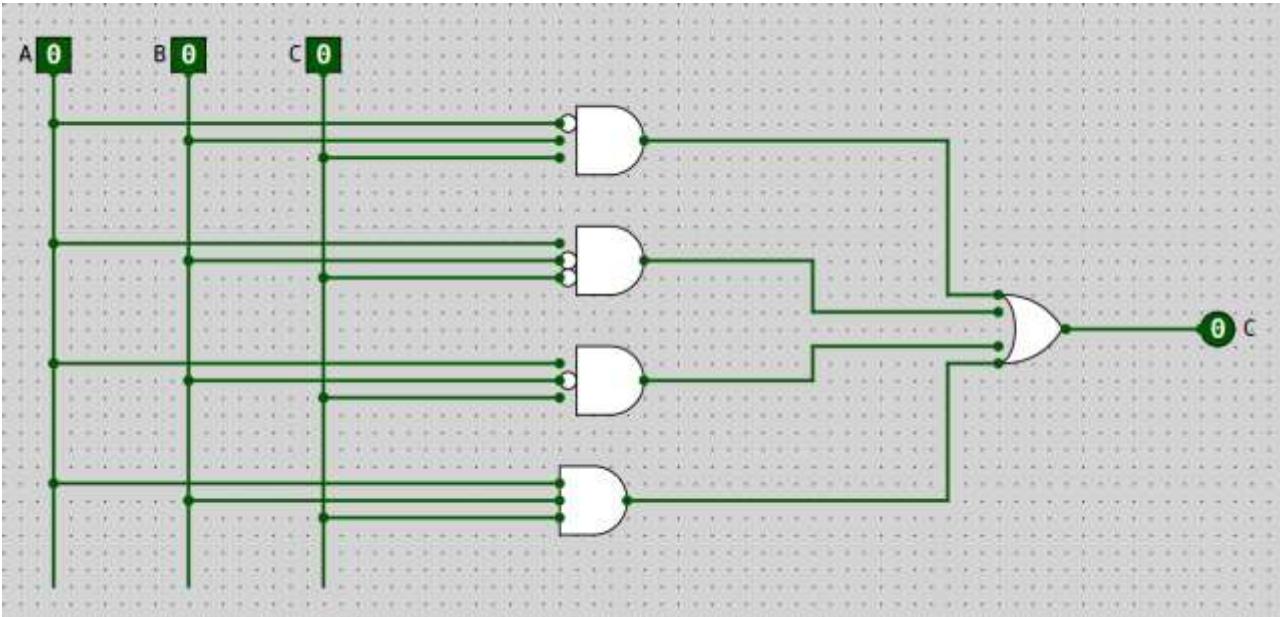


# Logic Conversion

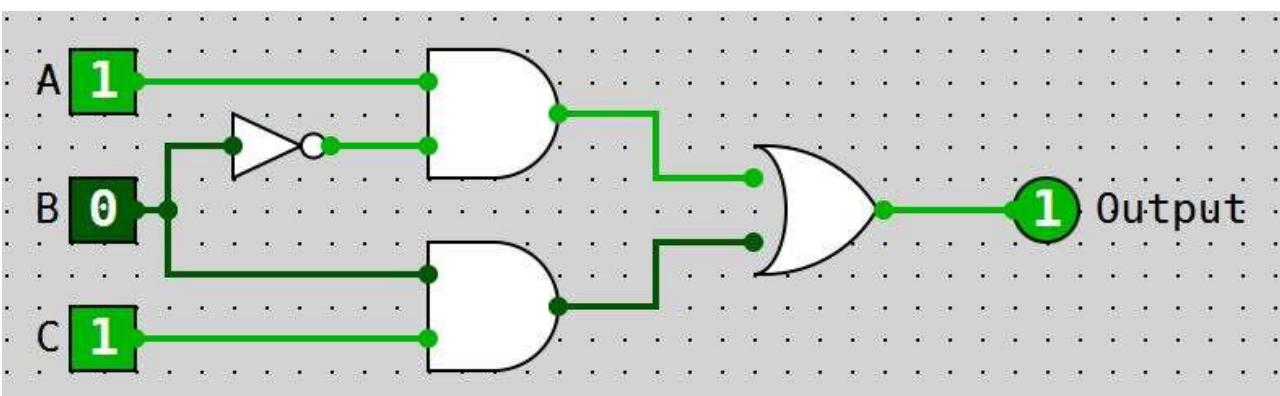
| A | B | C | Func<br>(A,B,C) |
|---|---|---|-----------------|
| 0 | 0 | 0 | X               |
| 0 | 0 | 1 | 0               |
| 0 | 1 | 0 | 0               |
| 0 | 1 | 1 | 1               |
| 1 | 0 | 0 | 1               |
| 1 | 0 | 1 | 1               |
| 1 | 1 | 0 | 0               |
| 1 | 1 | 1 | 1               |

Sum of products expressions are long and unwieldy, and they require lots of logic gates.

How can we get from the top circuit to the bottom one?



TO



---

## Karnaugh Maps

- A method of simplifying Boolean expressions by grouping together related terms
- Results in the simplest sum-of-products expression possible
- Allows for “don’t care” outputs

---

## Step 1: Create the K-Map

| A | B | C | Func(A,B,C) |
|---|---|---|-------------|
| 0 | 0 | 0 | X           |
| 0 | 0 | 1 | 0           |
| 0 | 1 | 0 | 0           |
| 0 | 1 | 1 | 1           |
| 1 | 0 | 0 | 1           |
| 1 | 0 | 1 | 1           |
| 1 | 1 | 0 | 0           |
| 1 | 1 | 1 | 1           |

- Distribute variables across the rows and columns using Gray code order
- Fill in corresponding entries

|    | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  |    |     |      |     |
| C' |    |     |      |     |

|    | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 1   |
| C' | 0  | 1   | X    | 0   |

---

## Step 2: Make Groupings

|           | <b>AB</b> | <b>AB'</b> | <b>A'B'</b> | <b>A'B</b> |
|-----------|-----------|------------|-------------|------------|
| <b>C</b>  | 1         | 1          | 0           | 1          |
| <b>C'</b> | 0         | 1          | X           | 0          |

|           | <b>AB</b> | <b>AB'</b> | <b>A'B'</b> | <b>A'B</b> |
|-----------|-----------|------------|-------------|------------|
| <b>C</b>  | 1         | 1          | 0           | 1          |
| <b>C'</b> | 0         | 1          | X           | 0          |

**Rules of K-map grouping:**

1. Groups must be rectangular (but they may wrap around edges!)
2. Groups may only contain “1”s or “X”s
3. All “1”s must be contained within at least one group
4. Groups must be as large as possible
5. The size of a group must be a power of 2 – note that you may have a group of size 1

## Step 3: Write Simplified Expression

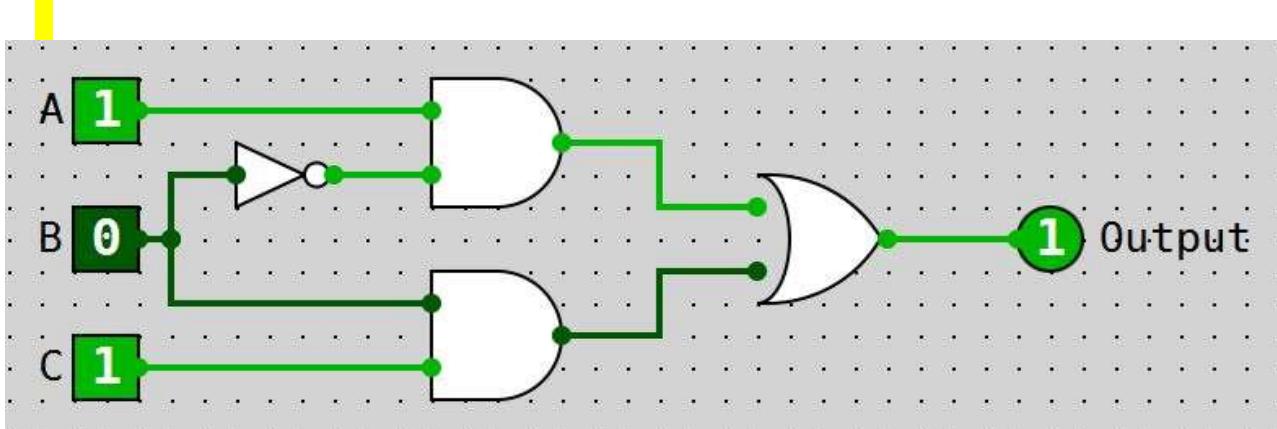
|    | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 1   |
| C' | 0  | 1   | X    | 0   |

|    | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 1   |
| C' | 0  | 1   | X    | 0   |

Pull out the simplified expression based on the K-map groupings

$$(ABC + A'BC) + (AB'C + AB'C')$$

$$BC + AB'$$



---

# K-Map Reducing Practice

| Q1 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 0   | 1    | 1   |
| C' | X  | 0   | 0    | 1   |

| Q2 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 0   |
| C' | 0  | 0   | 1    | 1   |

| Q3 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 0  | 1   | 0    | 1   |
| C' | 1  | 0   | 1    | 0   |

| Q4 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 1    | 1   |
| C' | 1  | 1   | 0    | 0   |

# K-Map Reducing Practice

| Q1 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 0   | 1    | 1   |
| C' | X  | 0   | 0    | 1   |

overlap

| Q2 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 0   |
| C' | 0  | 0   | 1    | 1   |

| Q3 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 0  | 1   | 0    | 1   |
| C' | 1  | 0   | 1    | 0   |

overlap

| Q4 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 1    | 1   |
| C' | 1  | 1   | 0    | 0   |

---

# K-Map Reducing Practice

| Q1 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 0   | 1    | 1   |
| C' | X  | 0   | 0    | 1   |

$$B + AC$$

| Q2 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 0    | 0   |
| C' | 0  | 0   | 1    | 1   |

$$AC + A'C'$$

| Q3 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 0  | 1   | 0    | 1   |
| C' | 1  | 0   | 1    | 0   |

$$ABC' + ABC + A'B'C' + A'BC$$

Impossible to simplify using sum-of-products!

| Q4 | AB | AB' | A'B' | A'B |
|----|----|-----|------|-----|
| C  | 1  | 1   | 1    | 1   |
| C' | 1  | 1   | 0    | 0   |

$$C + A$$

---

# CS 2110 - Lab 05

## Sequential Logic

Wednesday, September 15, 2021



---

## Homework 2

- Released!
- Due Wednesday, September 15<sup>th</sup> at 11:59 PM
  - Standard grace period until 9/16 with 25% deduction applies
- Files available on Canvas
- Submit on Gradescope
  - Double check that your grade on Gradescope is your desired grade!

---

## Homework 3

- Will release on Thursday, September 16<sup>th</sup>
- Due Wednesday, September 22<sup>nd</sup> at 11:59 pm
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 9/23 applies

---

## Timed Lab 1

- Will be on Monday, September 20<sup>th</sup>
- Full class period available
- Will cover HW2 Material primarily
  - Open book, open note, open internet (you can even look at homework files)
  - Please make sure Docker is working before your lab period begins.

---

## A Note on CircuitSim

- Do not use versions of CircuitSim you find out on the Internet. They are incompatible with our autograders and will corrupt your files.
- Seriously, don't do it.
- Use CircuitSim in Docker (recommended) or the JAR available in Canvas (requires JavaFX)

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 05
- 3) Access code: flip-flop
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



---

# Today's Topics

- Sequential Logic
  - What is sequential logic?
  - Level-triggered vs edge-triggered logic
  - RS Latches, Gated D-Latches and D Flip-Flops
- Intro to State Machines
  - (feat: **Dave the Funky Dancing Robot Crab**)
- Address Space and Addressability

---

## Combinational vs Sequential Logic

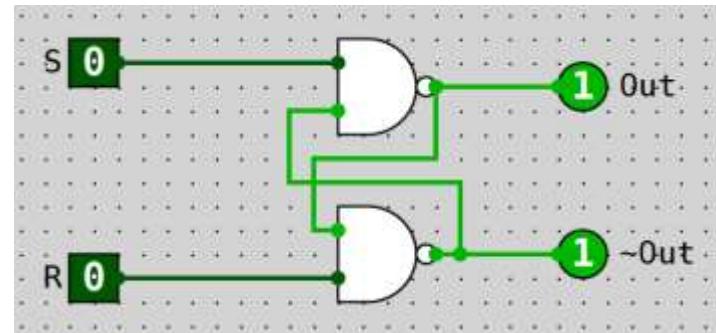
Combinational Logic: the outputs of the circuit depend only on the **current** combination of inputs. Examples?

Sequential Logic: output of the circuit depends on both the **current** inputs and **previous** values of the inputs. We will see some examples now.

---

## RS Latch

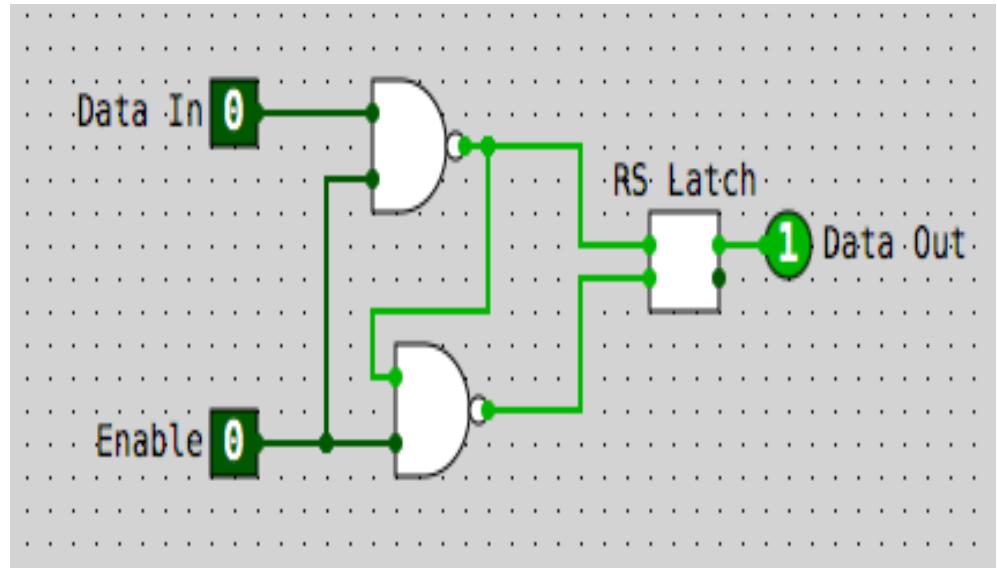
- The Basic Circuit of Sequential Logic
- Stores 1 bit
- 3 valid states:
  - $R = 1, S = 1$ : output at this state depends on previous state
  - $R = 1, S = 0$ : what is the output?
  - $R = 0, S = 1$ : what is the output?



---

## Gated D-Latch

- Essentially just an RS Latch with extra control
- Can choose when to save the output and what to save it as



---

## Level-Triggered Logic

- The previous two circuits are examples of level-triggered circuits.
- In level-triggered logic, the output can only change when the enable bit is 1
- When the enable bit is set to 0, then the output is unaffected by changes in the input



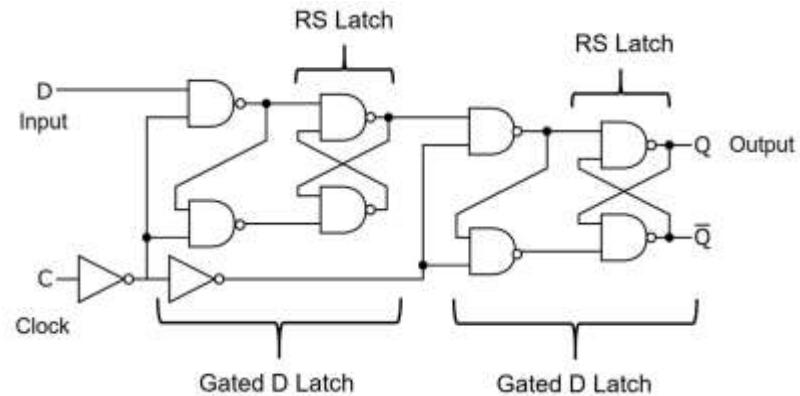
## Edge-Triggered Logic

- Many sequential logic circuits are based on clocks instead of enable bits
- Rising-edge triggered logic: The output can only change when the clock changes from 0 to 1
- Falling-edge triggered logic: The output can only change when the clock changes from 1 to 0

---

## D Flip-Flops

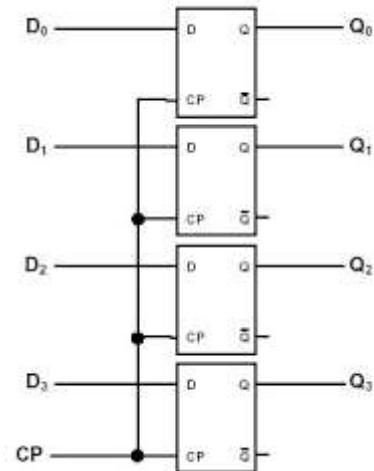
Rising edge-triggered! The left D-latch updates when the clock is 0, and the right D-latch updates when it is 1.



---

# Registers

- We can combine multiple latches to create a *register* to store multiple bits
- Essentially the same as a D Flip-Flop, but there are  $n$ -bit input or output lines instead of just one



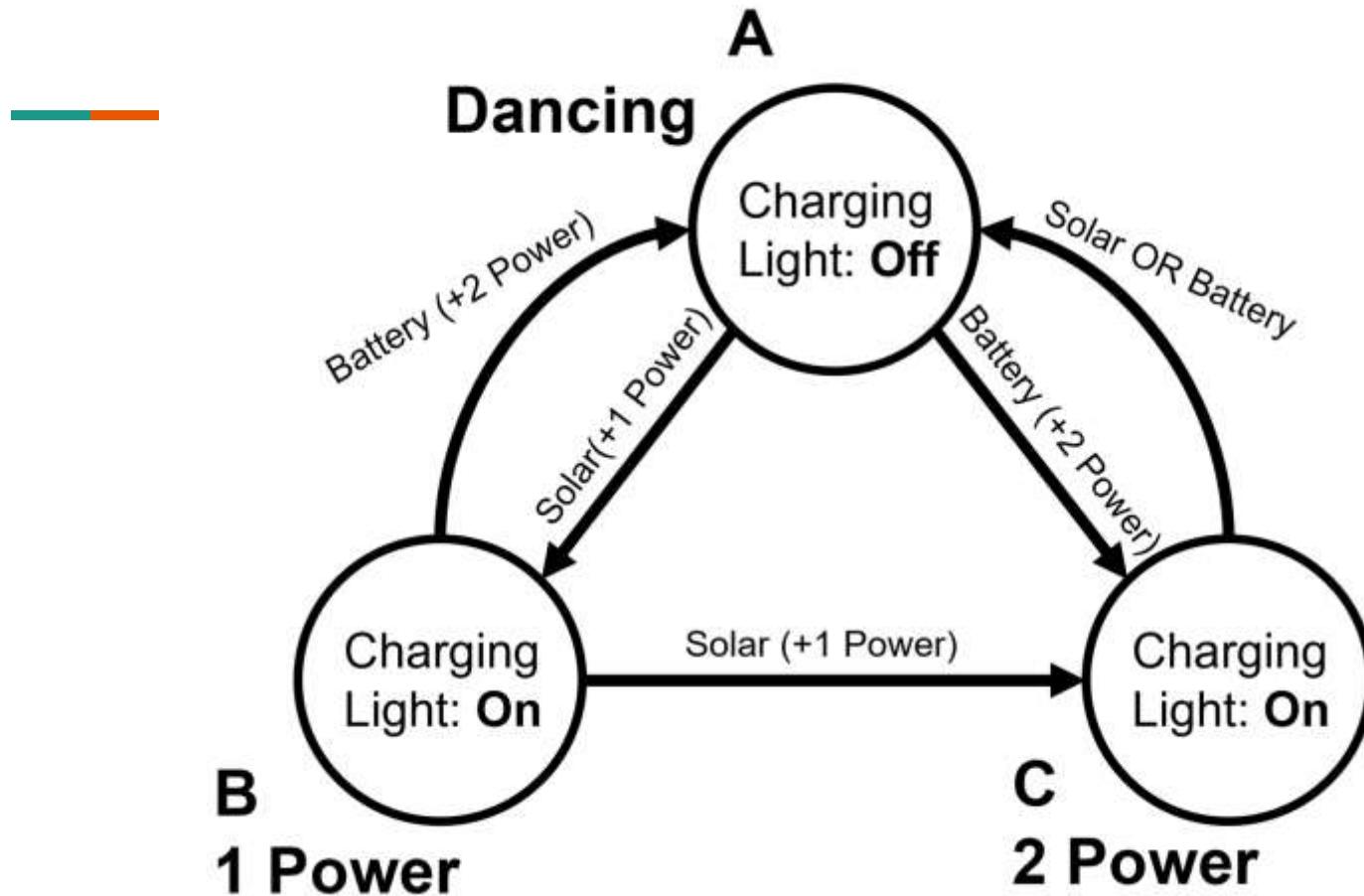
---

## State Machines!

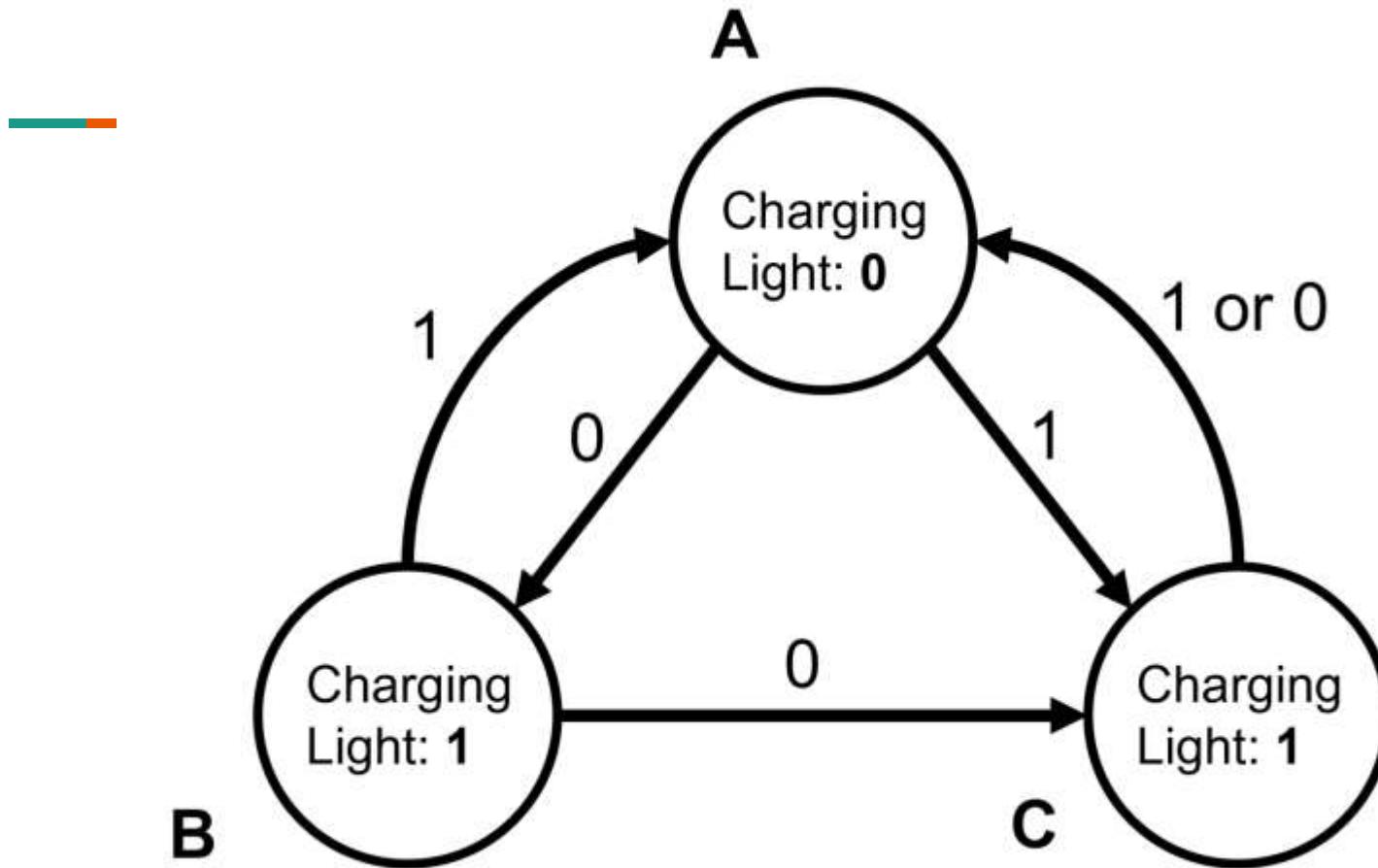
- Example: Dave, the Funky Dancing Robot Crab
  - Dave is a dancing robot crab. Before it can dance it must fully recharge (3 charges). It can recharge slow by solar energy (1 charge) or fast with a battery (2 charges).
  - Dave has 3 possible states:
    1. Dancing (requires 3 charges)
    2. Charging with 1 charge
    3. Charging with 2 charges



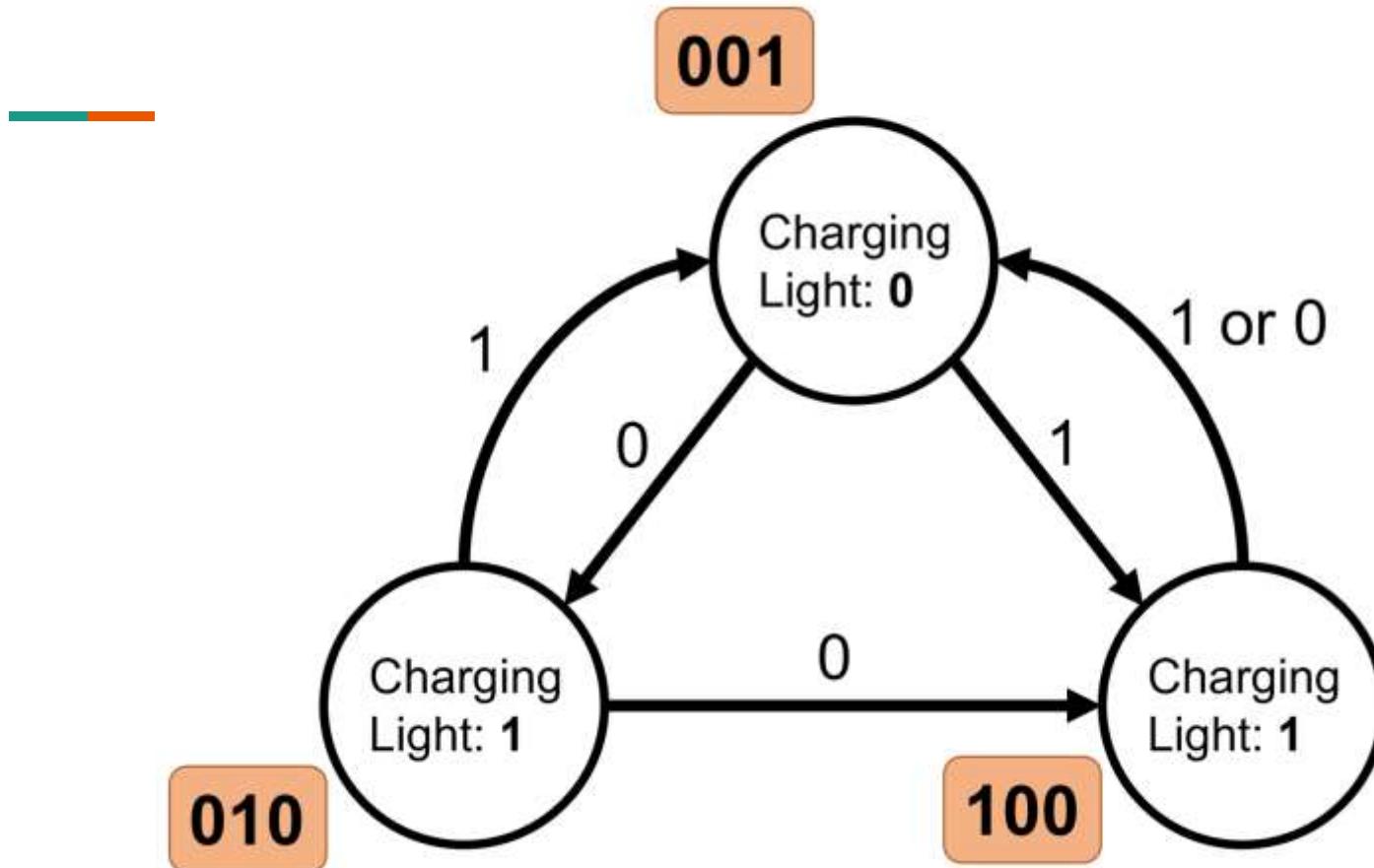
# Dave's State Transition Diagram



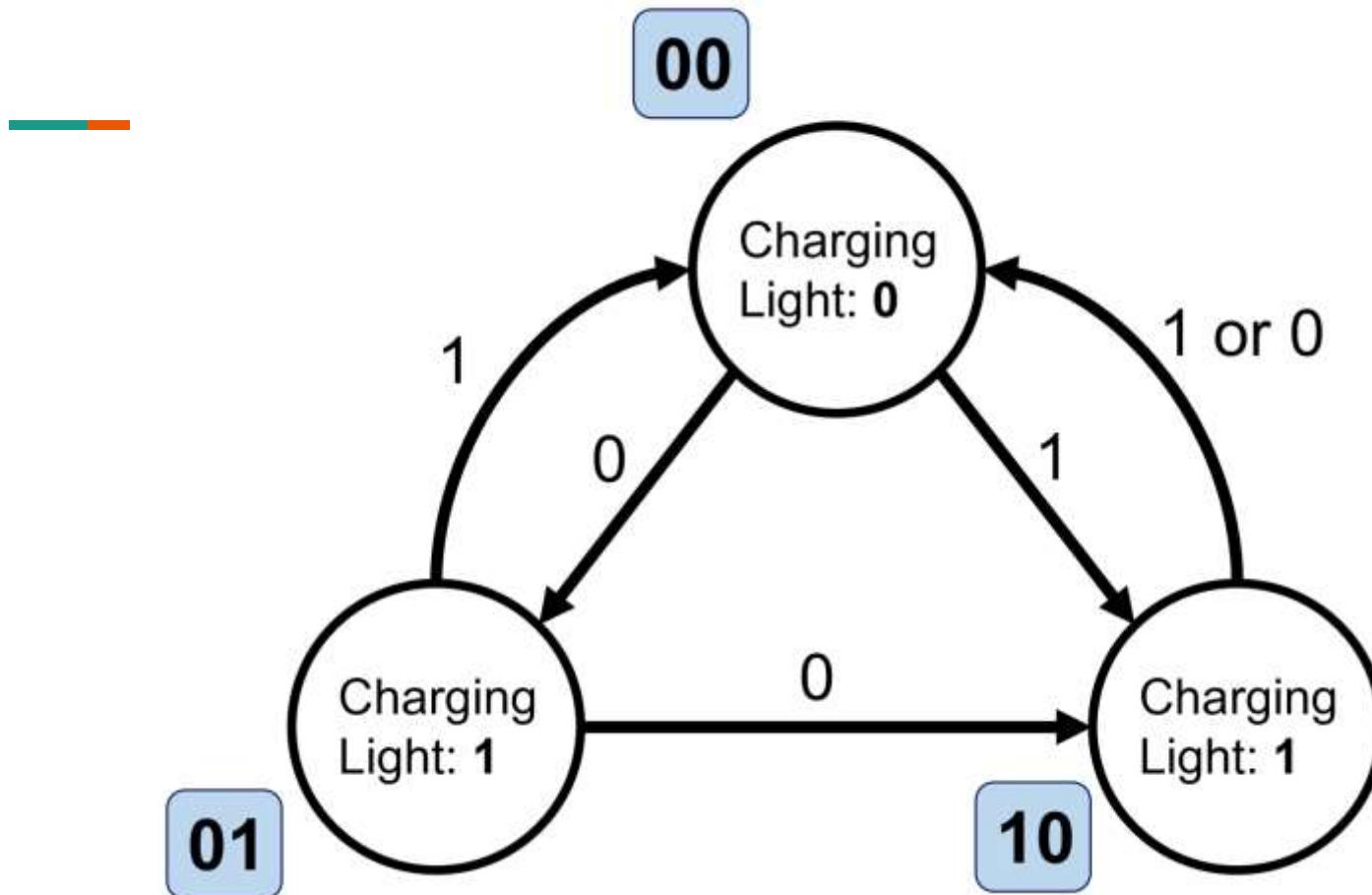
# State Transition Diagram - Representation



# State Machine Diagram - One Hot



# State Machine Diagram - Binary Encoded



---

## Address Space & Addressability

- Computer memory is made up of distinct *memory addresses*, each containing some amount of data
- Addressability: the amount of data stored at any given memory address
- Address space: the number of memory addresses that exist
  - An  $n$ -bit address line can represent  $2^n$  memory addresses

---

## Address Space & Addressability

- If a computer uses 16-bit lines to represent its memory addresses, what is its address space?
- What is the addressability of a system with 16 KiB of memory, composed of 2048 memory addresses?

---

# CS 2110 - Lab 06

## Intro to LC-3

Wednesday, September 22, 2021



---

## Homework 3

- State Machines!
- Released on Thursday, September 16<sup>th</sup>
- **Due Wednesday, September 22<sup>nd</sup> at 11:59 pm**
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 9/23 applies

---

## Homework 4

- The LC-3 datapath (more on this today!)
- Released on Thursday, September 23<sup>th</sup>
- **Due Wednesday, September 29<sup>nd</sup> at 11:59 pm**
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 9/30 applies
- Demoed! More details on this later :)

---

## Quiz 2

- Next Monday (September 27<sup>th</sup>)
- You must come to class and take the quiz here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TAs
- Full 75 minutes to take the quiz!
- A topic list will be posted on Canvas

---

## A note on Docker

- You *need* to have Docker working.
- Please come to office hours if you're still having regular issues (crashes, etc.)
- You will have issues on future timed labs if your Docker still isn't working correctly.

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 06
- 3) Access code: Tristate
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



---

# State Machines

- At any point, we are in one state, which defines our current outputs
- Even a simple computer represents a very complex state machine
  - A processor is always doing one of many things
  - There are lots of inputs (keyboard, mouse, power button, etc.)
- Regardless, the same fundamentals you learned in HW3 can be used to build a computer

---

## Reminder!

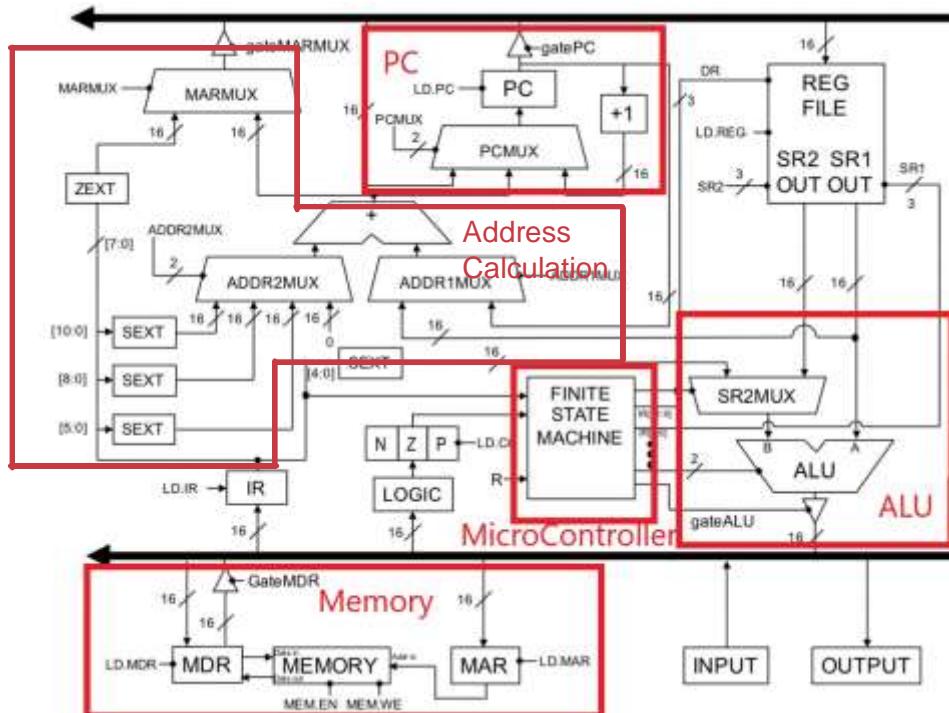
- State machines are NOT event-driven; they are locked to the clock
- The state is always updated once per clock cycle, even if you transition into the same state
- This is true of all sequential logic in modern computers

---

## The LC-3

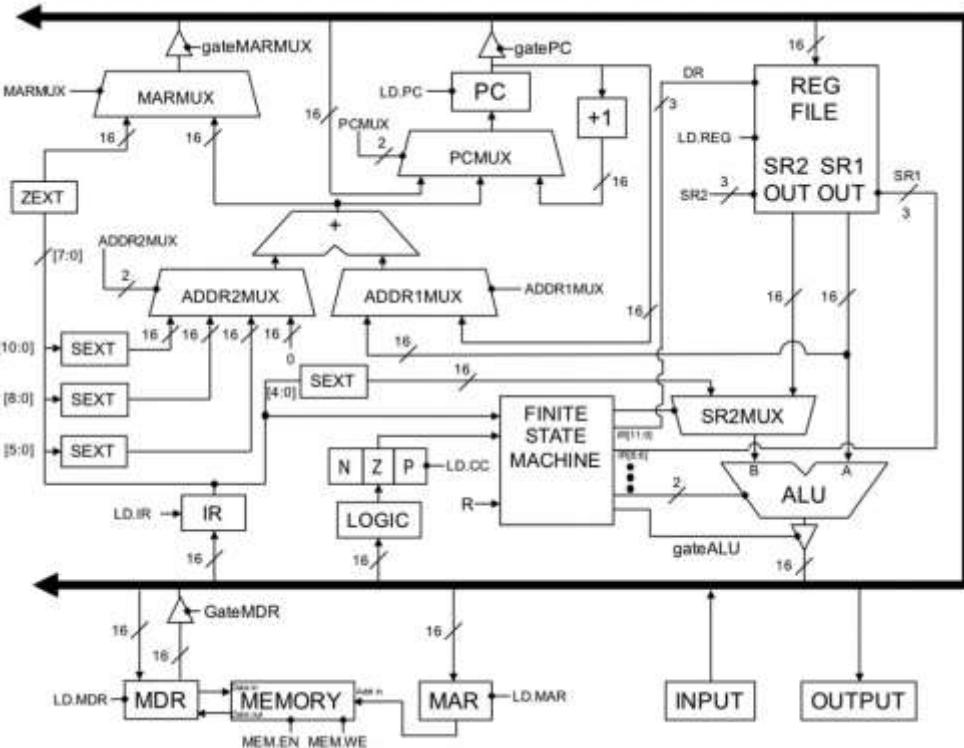
- A pedagogical processor architecture
- Combines everything we've learned so far
  - ALU, state machine, registers, memory, etc.
- Looks intimidating, but we'll break it down into its components

# LC-3 Components



# The Bus

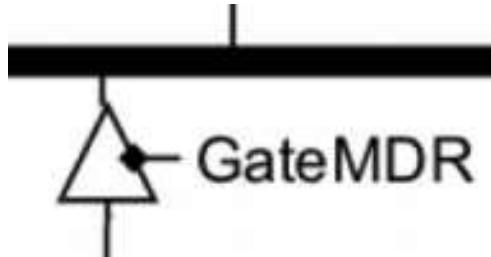
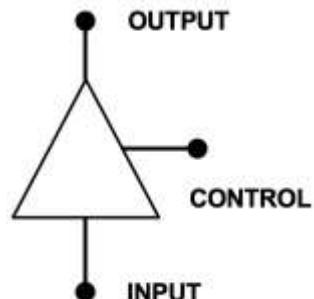
- 16-bit wire that transfers data between many components
- Only **one** value on wire at a time—why?



---

## Tristate Buffer

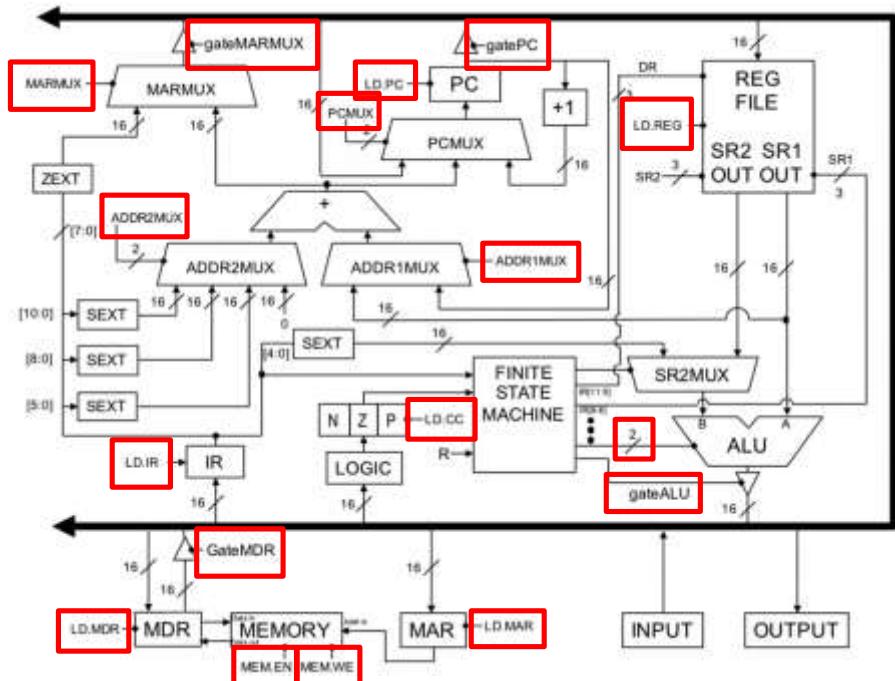
- Helps in making sure there is only one value on bus
- Allows us to choose whether or not to write a value



| CONTROL | INPUT | OUTPUT |
|---------|-------|--------|
| 0       | 0     | Z      |
| 0       | 1     | Z      |
| 1       | 0     | 0      |
| 1       | 1     | 1      |

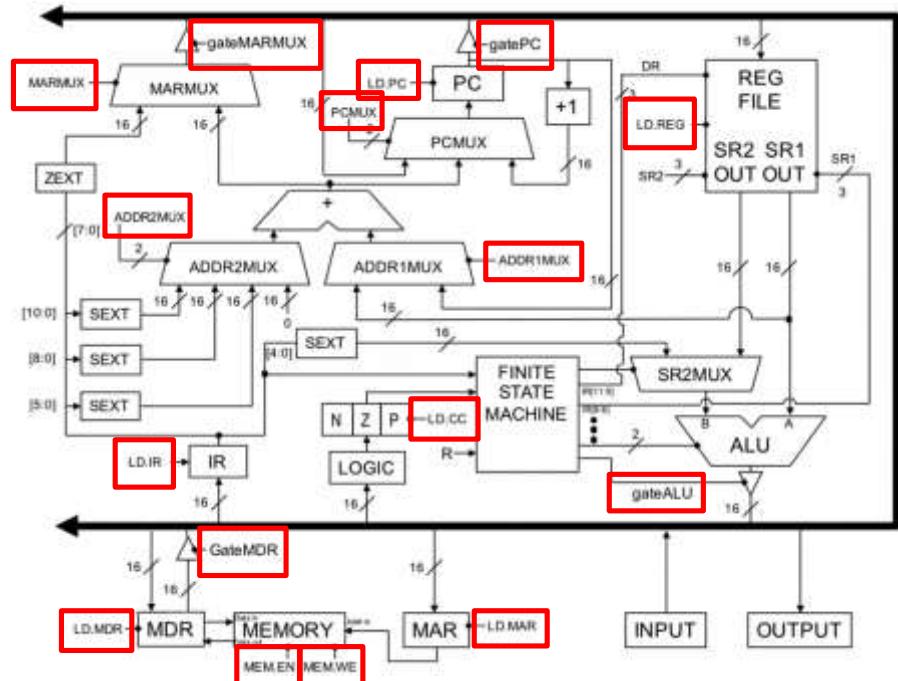
# Control Signals

- Used to move data around
  - LD. \_\_\_\_ – write enable for a register
  - gate\_\_\_\_ – tri-state buffer that allows data to go onto the bus
  - \_\_\_\_MUX – selector for a multiplexer



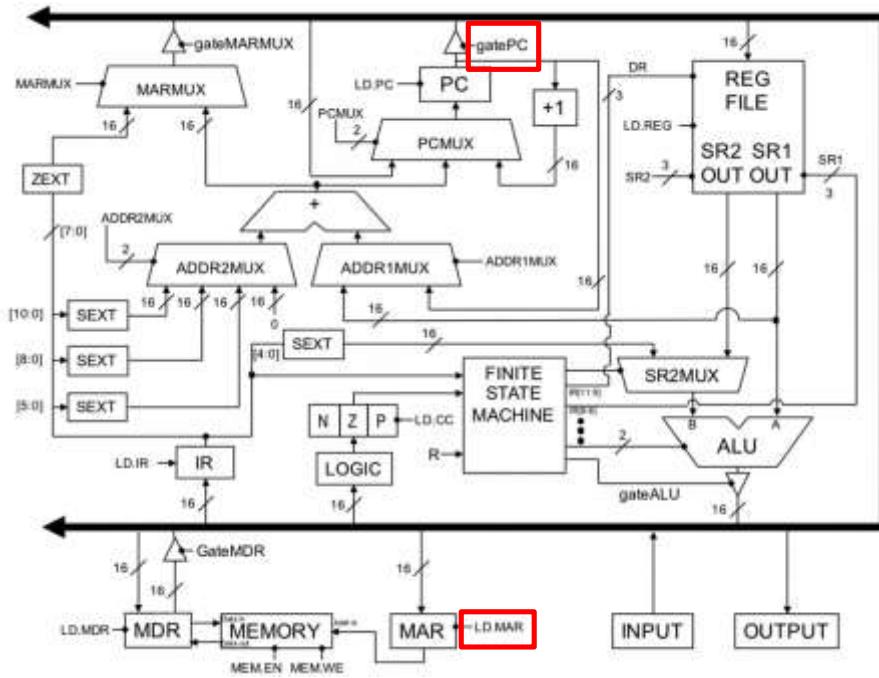
# Example

- Say we want to put the data from the PC (a register) into the MAR (another register). What signals do we turn on?



# Answer

- gatePC to drive the data in the PC onto the bus
- LD.MAR to enable writing into the MAR
- On the next clock edge, the data will be saved!



---

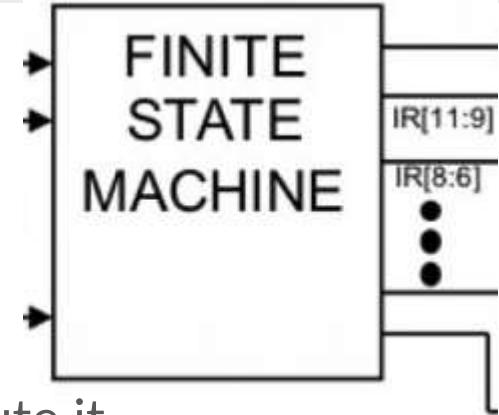
# Clock

- Clock cycles
  - Alternates between 0/1's
- One clock is connected to all the registers, keeping them in sync
- Ensure that we don't run into short circuits
- When a register has its Write Enable on, the next clock edge saves its current input

---

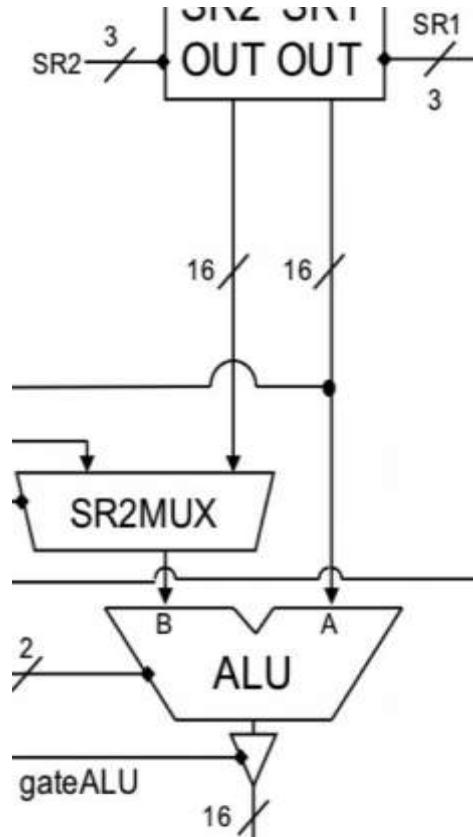
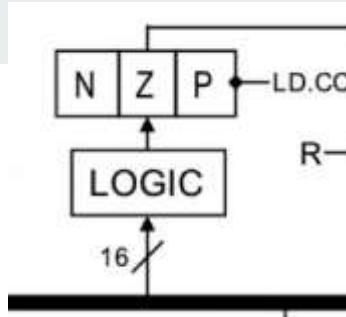
# The Microcontroller

- Finite state machine (FSM)
- Every instruction has a series of states to execute it
- Like in HW3, each state has
  - A set of outputs (control signals like ALUK, LdReg, etc.)
  - The next state to go to
- Ensures that correct operations take place, while preventing short circuits
- You'll be filling in parts of the state machine truth table for HW4



# ALU

- Two inputs:
  - SR1: Data from register
  - SR2: Data from register **OR** immediate value from the instruction
- Selector bits come from Finite State Machine
  - 00 → A + B; 01 → A AND B; 10 → NOT A; 11 → PASS A
- Drives output onto bus
- Condition codes (CC)
  - Set when writing to general purpose registers, LD.CC flag
  - Only three possible states: 001, 010, 100
  - What do they each mean?
- Does anything on this slide look familiar?



# What is Memory? - Registers VS. Memory (RAM)

---

You have been exposed to sequential logic. (i.e. registers)

**Registers:** Used for quick data storage on the processor.

---

**Random Access Memory:**

- Memory is like a really big array
- Contains instructions for programs that are currently being executed.
- Stores any data a program may need.

**The two “sides” of memory:**

- Address side
- Data side (or Data @ Address)

**NOTE: Everything here is  
technically in binary.**

# Memory Layout Example

This example is used to show the type of data that can be found in memory.

## Addresses

## Data @ Addresses

0x3000

ADD R1, R2, R0

0x3001

HALT

0x3002

0b0010000100010000

0x3003

0x2110

...

...

0x300A

#8464

LC-3 Instructions:  
Piece of some  
program in memory.

Random Data in Memory:  
Data used by a program  
currently being executed.

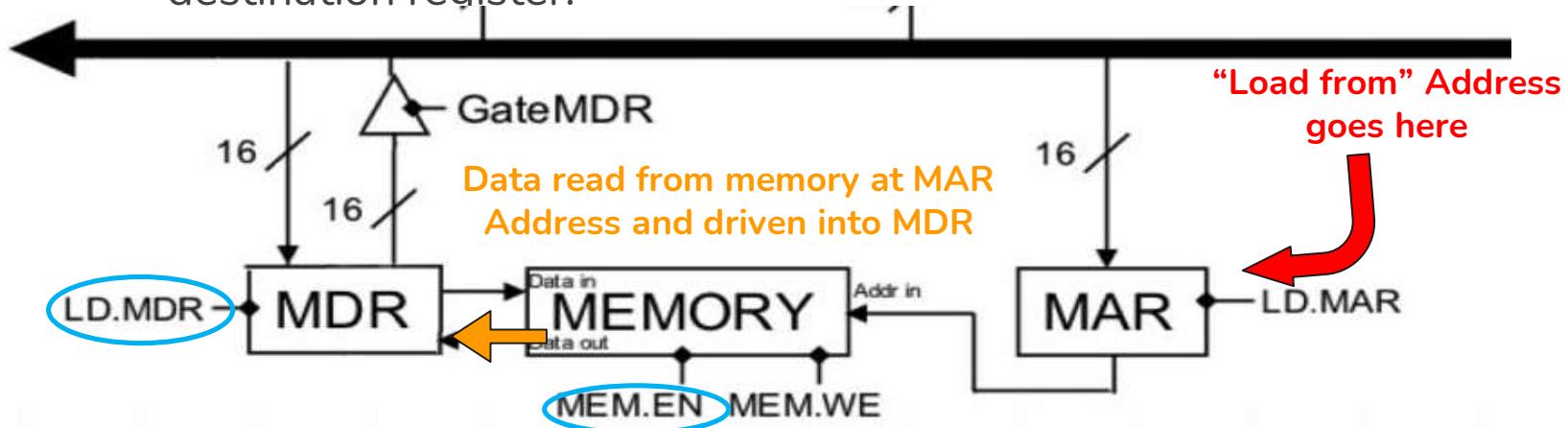
**OR**

Left over data from an old  
program previously executed.

# Loading from Memory

## Load data from memory:

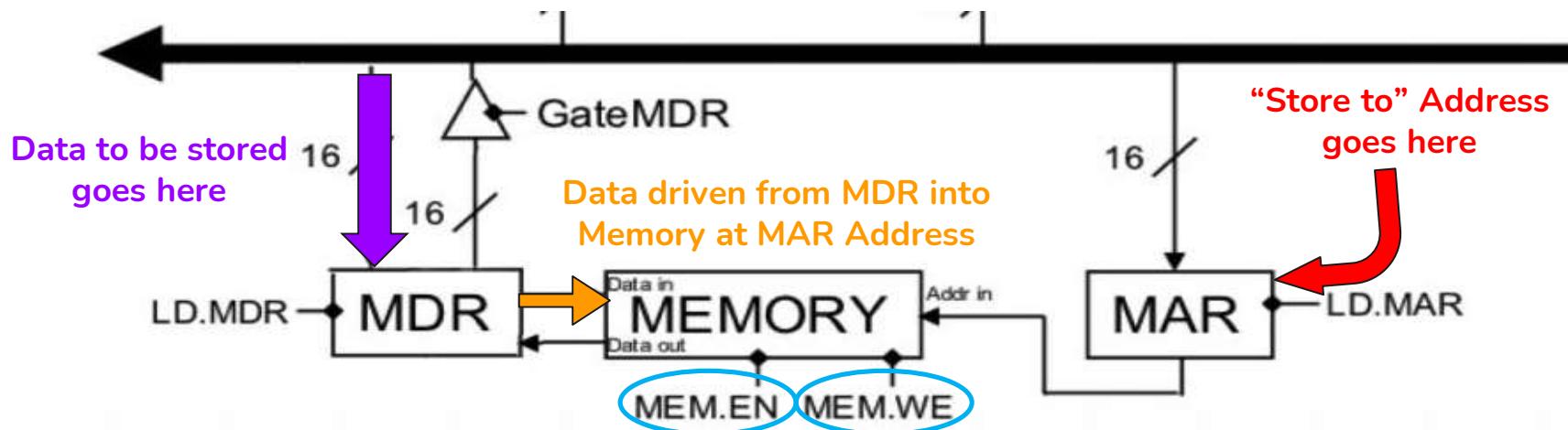
- Place the **address** of where the data is located in **MAR (Memory Address Register)**.
- Memory is **read** from that address and “driven” to the **MDR (Memory Data Register)**. After that, the data moves to its destination register.



# Storing into Memory

## Store data into memory:

- Data from a register is placed into the MDR.
- Address of where to store data is placed into the MAR.
- Data is stored in memory at the address specified by the MAR.



---

## Note

- Every piece of data in memory is just 16 bits of binary
- How does the LC-3 know...
  - Which addresses contain data that's an instruction?
  - Which addresses contain data that's an integer?
  - Which addresses contain data that's a character?
- Answer: it doesn't! You, the programmer, must keep track of what each piece of data means.

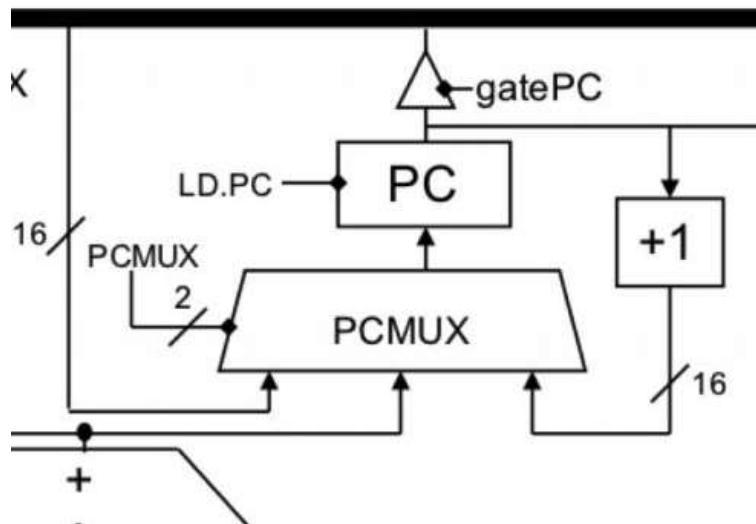
# LC3 Instruction Set

|     |      |    |     |   |    |           |
|-----|------|----|-----|---|----|-----------|
| ADD | 0001 | DR | SR1 | 0 | 00 | SR2       |
| ADD | 0001 | DR | SR1 | 1 |    | imm5      |
| AND | 0101 | DR | SR1 | 0 | 00 | SR2       |
| AND | 0101 | DR | SR1 | 1 |    | imm5      |
| BR  | 0000 | n  | z   | p |    | PCoffset9 |

|     |      |    |       |           |         |
|-----|------|----|-------|-----------|---------|
| LD  | 0010 | DR |       | PCoffset9 |         |
| LDI | 1010 | DR |       | PCoffset9 |         |
| LDR | 0110 | DR | BaseR |           | offset6 |
| LEA | 1110 | DR |       | PCoffset9 |         |
| NOT | 1001 | DR | SR    |           | 111111  |
| ST  | 0011 | SR |       | PCoffset9 |         |
| STI | 1011 | SR |       | PCoffset9 |         |
| STR | 0111 | SR | BaseR |           | offset6 |

# PC

- Holds the current program counter – address of next instruction to execute
- Repeat that: "the address of the *next* instruction to execute"
- It does NOT hold:
  - The current instruction
  - The address of the current instruction
- Three ways to update PC
  - PC + offset
  - PC + 1
  - Given a value for the PC via the bus
- The "correct" way is selected through PCMux

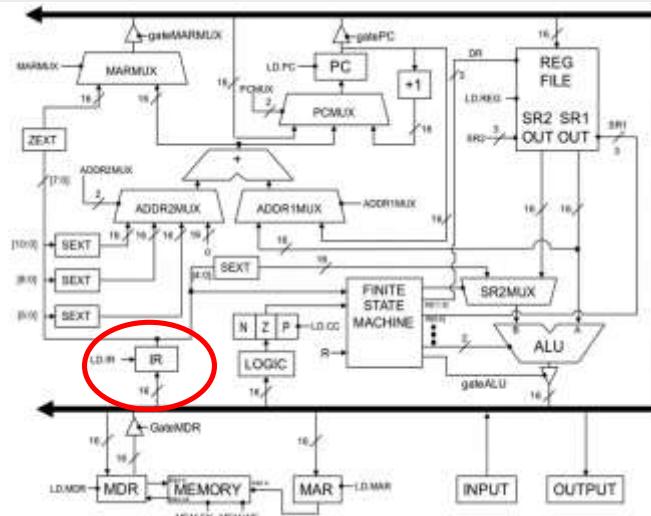


# IR

- Instruction register – holds the **value** of the **current** instruction
- For example, the instruction ADD R3, R1, R2 may look like:

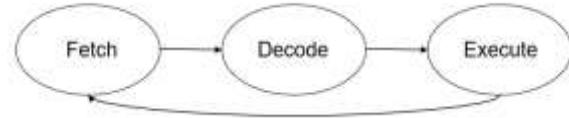
|     |      |     |     |   |    |     |
|-----|------|-----|-----|---|----|-----|
| ADD | 0001 | 011 | 001 | 0 | 00 | 010 |
|     | 0001 | DR  | SR1 | 0 | 00 | SR2 |

- Having the correct value in the IR helps the FSM determine the current state and send the correct control signals
- A new value can be read from the bus (with signal LD.IR) and the current value is directly connected to relevant components



# Executing Instructions

---



Instruction execution has three "stages", called *macrostates*

- Each macrostate is made of one or more *microstates* (1 per clock cycle)
- The microstates are the states in the microcontroller's state machine!

The macrostates are:

1. Fetch (3 clock cycles)
  - Load the next instruction from memory into the instruction register (IR)
  - Increment the PC
2. Decode (1 clock cycle)
  - The microcontroller looks at the instruction to figure out how to execute it
3. Execute (varies)
  - The microcontroller steps through several microstates and asserts the correct control signals to execute the instruction

**The book splits up executing instructions into seven phases. This is a more conceptual overview of how it works and is unrelated to the three macrostates.**

---

# CS 2110 - Lab 07

LC-3 Datapath

Wednesday, September 29, 2021



---

## Homework 4

- The LC-3 Datapath!
- Released on Thursday, September 23<sup>th</sup>
- **Due Thursday, September 30<sup>th</sup> at 11:59 pm**
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 10/1 applies

---

## Homework 4 Demo Logistics

- No homework assigned during demo week
  - Some office hours may be cancelled
- You may demo with any TA
- Sign up for a demo slot by **DATE** to be guaranteed
- Closed book, note, homework, etc.
- Not a quiz—we will prompt/guide you as needed
- Read Shawn's announcement for all the details

---

## Timed Lab 2

- Next Wednesday (October 6<sup>th</sup>)
- You must come to class and take the quiz here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TAs
- Full 75 minutes to complete the timed lab!
- On HW3 topics (state machines)

---

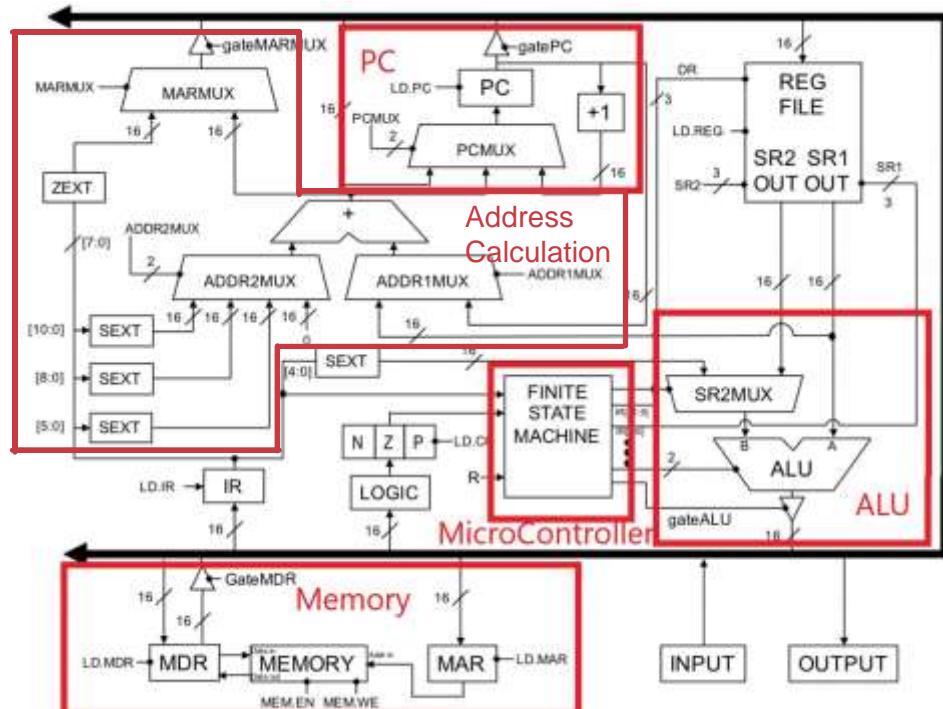
## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 07
- 3) Access code: NOP
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



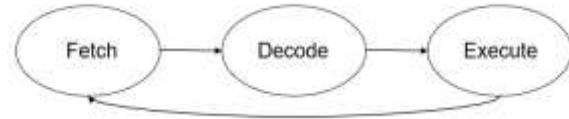
# LC-3 Components

(a quick review!)



# Executing Instructions

---



Instruction execution has three "stages", called *macrostates*

- o Each macrostate is made of one or more *microstates* (1 per clock cycle)
- o The microstates are the states in the microcontroller's state machine!

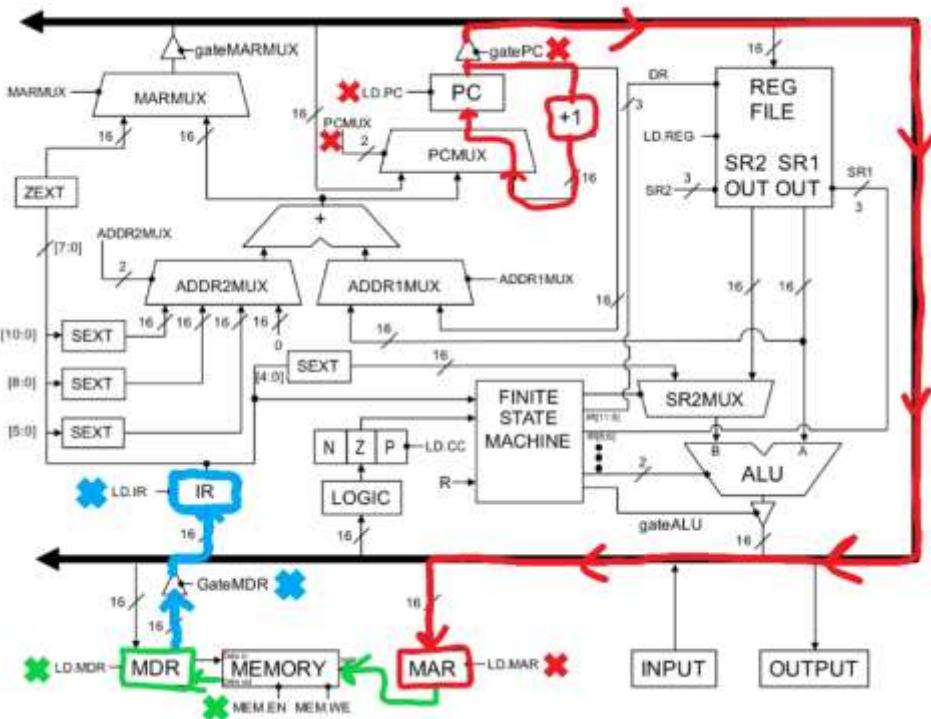
The macrostates are:

1. Fetch (3 clock cycles)
  - Load the next instruction from memory into the instruction register (IR)
  - Increment the PC
2. Decode (1 clock cycle)
  - The microcontroller looks at the instruction to figure out how to execute it
3. Execute (varies)
  - The microcontroller steps through several microstates and asserts the correct control signals to execute the instruction

When Execute is completed, the LC-3 starts over and fetches the next instruction.

# FETCH

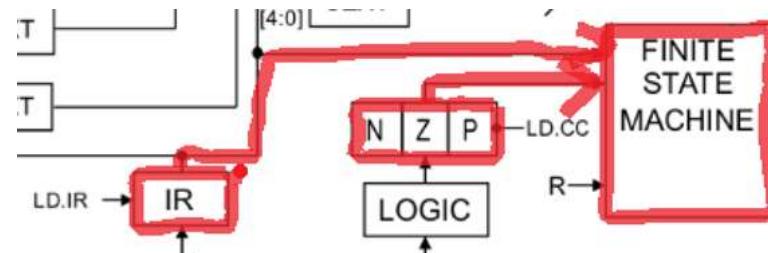
- Use PC value (address of next instruction) to read the next instruction from memory
- Load this value into the IR so that it can be decoded and executed later
- Increment the value of PC – it must always point to the **next** instruction to be run



---

## DECODE

- Now that the IR contains the instruction, the FSM can read the opcode.
- If the instruction is BR (branch), the FSM will also check the condition codes to ensure that the branching condition is met.
- The FSM will change to a specific state depending on the opcode. This takes a clock cycle in the LC-3, but some computers can combine DECODE with the last cycle of FETCH.



---

# EXECUTE

- Each instruction causes the EXECUTE phase to act differently.
- The EXECUTE macrostate takes a variable number of clock cycles; some instructions like ADD are quick, while others like LDI take longer.
- You have implemented the EXECUTE macrostate for many instructions (as well as the FETCH macrostate) in HW4

---

# LC-3 Assembly Instructions

---

# ADD

- The ADD instruction is opcode 0001.
- It has two formats (“regular” and immediate) which each work slightly differently.
- Bit 5 signals the format in use: immediate ADD if it is set, and “regular” two-register ADD if it is cleared
- ADD adds the two arguments and stores the result in the destination register (DR).
  - $DR = SR1 + SR2$
- ADD sets the condition codes.
- ADD instruction example: ADD R0, R1, R2

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

---

## ADD (immediate)

- The ADD instruction has a second version, which is enabled by setting bit 5 in the instruction: this bit is wired directly to the SR2MUX selector input
- This interprets the lower 5 bits as a 5-bit 2's complement number. It is sign-extended to 16 bits before the addition takes place.
  - $DR = SR + imm5$
- As a result, the range of numbers that can be supplied to this instruction is [-16,15]. To add larger numbers, either use a register to hold the second value, or use multiple immediate-add instructions.
- Immediate ADD example: ADD R0, R1, #5

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

---

# AND

- The AND instruction is opcode 0101.
- It has two formats as well and works in the exact same way as ADD except performing bitwise AND instead of ADD.
  - $DR = SR1 \And SR2$
- As with ADD, the result is stored in the destination register
- AND sets the condition codes.
- AND instruction example: AND R0, R1, R2

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 0  | 1  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

---

## AND (immediate)

- Like ADD, AND has a version that supports ANDing a register with a 5-bit 2's complement immediate value.
  - $DR = SR \& imm5$
- This is extremely useful when you need to store the value 0 in a register; any register ANDed with zero results in zero.
- Immediate AND instruction example: AND R3, R3, #0

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 1  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

---

# NOT

- The NOT instruction is opcode 1001.
- It only has one format as shown below.
- NOT performs bit-wise negation of the source register (SR) and stores the result in destination register (DR).
  - $DR = \sim SR$
- NOT sets the condition codes.
- NOT instruction example: NOT R1, R0
- Unlike ADD and AND, there is only one source register for NOT.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 1  | 0  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 |

---

## LD

- The LD instruction is opcode 0010. Remember: PC\* is the incremented PC.
- LD is the load instruction. It loads the value at the memory address  $PC^* + PCOffset9$  and puts it in the destination register.
  - $DR = \text{mem}[PC^* + PCOffset9]$
- PCOffset9 is a 9-bit 2's complement value
- LD sets the condition codes.
- LD R0, #0 reads the value at the location in memory addressed by  $PC^*$
- LD R0, #5 reads the value at the location in memory addressed by  $PC^* + 5$

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

---

## ST

Remember: PC\* is the incremented PC.

- The ST instruction is opcode 0011.
- ST is the store instruction and is very similar to LD. It stores the value in the source register at the memory address  $\text{PC}^* + \text{PCOffset9}$ .
  - $\text{mem}[\text{PC}^* + \text{PCOffset9}] = \text{SR}$
- ST R0, #5 stores the value in R0 at the memory address  $\text{PC}^* + 5$

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 0  | 1  | 1  | 1  | 1  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

---

# LDR

- The LDR instruction is opcode 0110.
- LDR is the Load Base+Offset instruction. It calculates its address by adding a register's value to a 6-bit 2's complement value that comes from the instruction.
  - $DR = \text{mem}[\text{BaseR} + \text{offset}_6]$
- LDR sets the condition codes.
- LDR instruction example: `LDR R0, R5, #0`
- LDR is useful when working with arrays. For example, if R5 were the address of the start of an array, the instruction would put the first element of the array into R0.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 1  | 1  | 0  | 1  | 0  | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |

---

# STR

- The STR instruction is opcode 0111.
- STR is the Store Base+Offset instruction.
- It stores the value from the source register at the memory location given by the sum of the base register's value and the immediate offset.
  - $\text{mem}[\text{BaseR} + \text{offset6}] = \text{SR}$
- Like LDR, STR is useful when working with arrays.
- STR instruction example: STR R0, R5, #6

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 0  | 1  | 1  | 1  | 0  | 0  | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 |

---

## LDI

Remember: PC\* is the incremented PC.

- The LDI instruction is opcode 1010.
- LDI is the Load Indirect instruction.
- It reads the value at the memory address  $PC^* + PCOffset9$ , then uses this value as the memory address to retrieve the final data.
  - $DR = \text{mem}[\text{mem}[PC^* + PCOffset9]]$
- PCOffset9 is a literal 9-bit 2's complement value.
- LDI sets the condition codes.

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 0  | 1  | 0  | 1  | 1  | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |

STI

**Remember: PC\* is the incremented PC.**

- The STI instruction is opcode 1011.
  - STI is the Store Indirect instruction.
  - It reads the value at the memory address  $PC^* + PCOffset9$ , then uses this value as the memory address at which it will store data retrieved from a general-purpose register.
    - $mem[mem[PC^* + PCOffset9]] = SR$
  - PCOffset9 is a literal 9-bit 2's complement value .

---

# LEA

Remember: PC\* is the incremented PC.

- The LEA instruction is opcode 1110.
- LEA is load effective address.
- It stores the value of  $PC^* + PCOffset9$  in the destination register.
  - $DR = PC^* + PCOffset9$
- LEA does **NOT** set the condition codes (as of version 3 of the textbook).
- LEA is great when you want to get the address of a data structure or array but read its value later.
- LEA instruction example: `LEA R0, #5`

|    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1  | 1  | 1  | 0  | 1  | 1  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

---

## BR

Remember: PC\* is the incremented PC.

- The BR instruction is opcode 0000.
- The BR instruction conditionally sets the PC to the value  $PC^* + PCOffset9$ , allowing you to branch to a different instruction rather than running the next one
- The "conditional part" is handled by only taking the branch if the value in CC is equal to any of the control bits set in the instruction. For example, if the NZP bits are 110 and the value in the CC registers is 001, then the branch will not be taken and the PC will be incremented by one like normal.
- The instructions BR and BRnzp both function as unconditional branches; they will always be taken.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 0  | 0  | 1  | 1  | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |

**NOP**

- When a BR instruction has a 0 in each of the NZP locations, it will never activate its branch condition.
  - For this reason, a Branch instruction composed of all 0s is considered a NOP (should be said as No Operation or No-Op).
  - Running this instruction will have no effect at all on the datapath, beside the ordinary incrementation of the PC during the FETCH stage.

---

## Which instructions set the condition codes?

- ALU instructions
  - ADD, AND, NOT
- Load Instructions
  - LD, LDR, LDI
  - Not LEA
- Basically, any instruction that updates the register file (except LEA)

# Labels

- Some LC-3 instructions use the PC-relative addressing mode for operands
- These accept labels as a second parameter, allowing programmers to load from labeled memory addresses.
- Labels are not stored anywhere in memory. Instead, the assembler calculates the right numerical offset to get to the desired label.
- To calculate the offset yourself, find the difference between the memory address of the label and the **incremented** address of the instruction (which will be the value stored in the PC when it executes).

| Instruction       | Address | PC*   |
|-------------------|---------|-------|
| LD R1, LABEL      | x3000   | x3001 |
| ...               | x3001   | x3002 |
| ...               | x3002   | x3003 |
| ...               | x3003   | x3004 |
| LABEL .fill #1234 | x3004   | x3005 |

$$\begin{aligned} \text{Address of LABEL} - \text{value of incremented PC} \\ = x3004 - x3001 \\ = 3 \end{aligned}$$

Therefore, LD R5, LABEL becomes LD R5, 3  
In machine code: [0010] [101] [000000011]

---

# LC-3 Assembly “Pseudo-Ops”

- Not actually instructions, but directions given to the assembler
- **.orig** – tells assembler to put block of code at desired address
  - `.orig x3000`
- **.stringz** – assembler will put a string of characters in memory followed by ‘\0’
  - `.stringz "something"`
- **.blkw** – allocates memory for specified label
  - `.blkw 10`
- **.fill** – puts value in memory location
  - `.fill x4040`
- **.end** – denotes end of block, closing tag for **.orig**

---

# Pseudo-Ops Example

| Instructions |                       |
|--------------|-----------------------|
| .orig        | x3000                 |
| string_label | .stringz "Hey there!" |
| block_label  | .blkw 5               |
| fill_label   | .fill x1234           |
| .end         |                       |

| Address |             |
|---------|-------------|
| x2FFE   | Don't know  |
| x2FFF   |             |
| x3000   | "H"         |
| x3001   | "e"         |
| x3002   | "y"         |
| x3003   | " "         |
| x3004   | "t"         |
| x3005   | "h"         |
| x3006   | "e"         |
| x3007   | "r"         |
| x3008   | "e"         |
| x3009   | "!"         |
| x300A   | 0 (NULL)    |
| x300B   |             |
| x300C   |             |
| x300D   | block_label |
| x300E   |             |
| x300F   |             |
| x3010   | x1234       |
| x3011   | Don't know  |
| x3012   |             |
| x3013   | either      |

# LC-3 Assembly File Layout

```
.orig x3000
 LD R1, B
 LD R0, A
 HALT
A .fill 2
ARRAY .fill x6000
.end
.orig x6000
 .fill 1
 .fill 2
 .fill 3
.end
```



|       | Address |            |
|-------|---------|------------|
|       | xFFE    | Don't know |
|       | xFFF    |            |
|       | x3000   | LD R1, B   |
|       | x3001   | LD R0, A   |
|       | x3002   | HALT       |
| A     | x3003   | 2          |
| ARRAY | x3004   | x6000      |
|       | x3005   | Don't know |
|       |         |            |
|       | x5FE    |            |
|       | x5FF    | Don't know |
|       | x6000   | 1          |
|       | x6001   | 2          |
|       | x6002   | 3          |
|       | x6003   |            |
|       | x6004   | Don't know |

---

## Assembly / Complx Example

- Given two inputs (at labels A and B), calculate A OR B and store it at the label RESULT
  - LC-3 doesn't have an OR instruction... how can we calculate this?
  - What instructions can we use to read from or write to memory at a label?

---

# CS 2110 - Lab 08

LC-3 Assembly Debugging  
with Complx

Monday, October 4, 2021



---

## Homework 5

- Basic assembly programming
- Released on Thursday, October 7<sup>th</sup>
- **Due Wednesday, October 13<sup>th</sup> at 11:59 pm**
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 10/14 applies

---

## Timed Lab 2

- This Wednesday (October 6<sup>th</sup>)
- You must come to class and take the TL here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TAs
- Full 75 minutes to complete the timed lab!
- On HW3 topics (state machines)

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 08
- 3) Access code: **Breakpoint**
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



# Review: LC-3 Instructions

|     |      |    |     |   |    |     |
|-----|------|----|-----|---|----|-----|
| ADD | 0001 | DR | SR1 | 0 | 00 | SR2 |
|-----|------|----|-----|---|----|-----|

|     |      |    |     |   |      |  |
|-----|------|----|-----|---|------|--|
| ADD | 0001 | DR | SR1 | 1 | imm5 |  |
|-----|------|----|-----|---|------|--|

|     |      |    |     |   |    |     |
|-----|------|----|-----|---|----|-----|
| AND | 0101 | DR | SR1 | 0 | 00 | SR2 |
|-----|------|----|-----|---|----|-----|

|     |      |    |     |   |      |  |
|-----|------|----|-----|---|------|--|
| AND | 0101 | DR | SR1 | 1 | imm5 |  |
|-----|------|----|-----|---|------|--|

|    |      |   |   |   |  |           |
|----|------|---|---|---|--|-----------|
| BR | 0000 | n | z | p |  | PCoffset9 |
|----|------|---|---|---|--|-----------|

|     |      |     |       |        |  |  |
|-----|------|-----|-------|--------|--|--|
| JMP | 1100 | 000 | BaseR | 000000 |  |  |
|-----|------|-----|-------|--------|--|--|

|     |      |   |  |  |            |  |
|-----|------|---|--|--|------------|--|
| JSR | 0100 | 1 |  |  | PCoffset11 |  |
|-----|------|---|--|--|------------|--|

|      |      |   |    |       |        |  |
|------|------|---|----|-------|--------|--|
| JSRR | 0100 | 0 | 00 | BaseR | 000000 |  |
|------|------|---|----|-------|--------|--|

|    |      |    |  |  |           |  |
|----|------|----|--|--|-----------|--|
| LD | 0010 | DR |  |  | PCoffset9 |  |
|----|------|----|--|--|-----------|--|

|     |      |    |  |           |  |  |
|-----|------|----|--|-----------|--|--|
| LDI | 1010 | DR |  | PCoffset9 |  |  |
|-----|------|----|--|-----------|--|--|

|     |      |    |       |         |  |  |
|-----|------|----|-------|---------|--|--|
| LDR | 0110 | DR | BaseR | offset6 |  |  |
|-----|------|----|-------|---------|--|--|

|     |      |    |  |           |  |  |
|-----|------|----|--|-----------|--|--|
| LEA | 1110 | DR |  | PCoffset9 |  |  |
|-----|------|----|--|-----------|--|--|

|     |      |    |    |        |  |  |
|-----|------|----|----|--------|--|--|
| NOT | 1001 | DR | SR | 111111 |  |  |
|-----|------|----|----|--------|--|--|

|    |      |    |  |           |  |  |
|----|------|----|--|-----------|--|--|
| ST | 0011 | SR |  | PCoffset9 |  |  |
|----|------|----|--|-----------|--|--|

|     |      |    |  |           |  |  |
|-----|------|----|--|-----------|--|--|
| STI | 1011 | SR |  | PCoffset9 |  |  |
|-----|------|----|--|-----------|--|--|

|     |      |    |       |         |  |  |
|-----|------|----|-------|---------|--|--|
| STR | 0111 | SR | BaseR | offset6 |  |  |
|-----|------|----|-------|---------|--|--|

|      |      |      |  |           |  |  |
|------|------|------|--|-----------|--|--|
| TRAP | 1111 | 0000 |  | trapvect8 |  |  |
|------|------|------|--|-----------|--|--|

Don't worry about JSR/JSRR/TRAP... for now

## Review: Labels

- PC-relative instructions accept labels instead of a PCOffset.
- Labels are not stored anywhere in memory—they only exist at assemble time
- To calculate the offset yourself, find the difference between the memory address of the label and the address of the PC when the instruction executes.
- Maximum range for PCOffset9? PCOffset11?

| Instruction       | Address | PC*   |
|-------------------|---------|-------|
| LD R1, LABEL      | x3000   | x3001 |
| ...               | x3001   | x3002 |
| ...               | x3002   | x3003 |
| ...               | x3003   | x3004 |
| LABEL .fill #1234 | x3004   | x3005 |

Address of LABEL – value of incremented PC  
= x3004 – x3001  
= 3

Therefore, LD R5, LABEL becomes LD R5, 3  
In machine code: [0010] [101] [000000011]

---

## Review: Pseudo-Ops

- Not actually instructions, but directions given to the assembler to tell it how to set up memory
- `.orig` – tells assembler to put block of code at desired address
  - `.orig x3000`
- `.stringz` – assembler will put a string of characters in memory followed by '\0'
  - `.stringz "something"`
- `.blkw` – allocates memory for specified label – very useful for arrays
  - `.blkw 10`
- `.fill` – puts value in memory location
  - `.fill x4040`
- `.end` – denotes end of block, closing tag for `.orig`

# LC-3 Assembly File Layout

```
.orig x3000
 LD R1, B
 LD R0, A
 HALT
A .fill 2
ARRAY .fill x6000
.end
.orig x6000
 .fill 1
 .fill 2
 .fill 3
.end
```



|       | Address |            |
|-------|---------|------------|
|       | xFFE    | Don't know |
|       | xFFF    |            |
|       | x3000   | LD R1, B   |
|       | x3001   | LD R0, A   |
|       | x3002   | HALT       |
| A     | x3003   | 2          |
| ARRAY | x3004   | x6000      |
|       | x3005   | Don't know |
|       |         |            |
|       | x5FE    |            |
|       | x5FF    | Don't know |
|       | x6000   | 1          |
|       | x6001   | 2          |
|       | x6002   | 3          |
|       | x6003   |            |
|       | x6004   | Don't know |

---

# Common Assembly Techniques

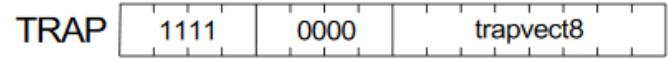
- Use a semicolon to write comments
- Clearing a register – `AND R1, R1, 0 ; R1 = R1 & 0`
  - You don't necessarily have to clear a register before you use it!
  - Example: `ADD R3, R2, R1` does not require you to clear R3 first
- If-statements and while-loops use BR statements to conditionally jump (follow the templates provided in lecture!)
  - E.g. `BRz` will branch when condition codes are 0
  - `BR` or `BRnzp` will branch unconditionally
- Setting condition codes – `ADD R1, R1, 0 ; R1 = R1 + 0`

---

# Common Assembly Techniques

- Subtraction/negation – use NOT (flip the bits) and AND (add 1)
  - `NOT R3, R1 ; R3 = ~R1`  
`ADD R3, R3, 1 ; R3 = ~R1+1 = -R1`  
`ADD R3, R2, R3 ; R3 = R2 + (-R1) = R2-R1`
- Copy a value to another register
  - `ADD R2, R1, 0 ; R2 = R1 + 0`

# TRAPs



- Subroutines built into LC-3 to simplify instructions
- Look like normal instructions, but are aliases for TRAP calls
  - Writing "HALT" is exactly the same as writing "TRAP x25"
- Each has a corresponding 8-bit Trap Vector.
  - Provides an index for the Trap Vector Table located at addresses x0000 to x00FF
  - Contains the address of the TRAP function

**HALT** (x25): stops running the program

**OUT** (x21): takes character (in ASCII) in R0 and prints it on console

**PUTS** (x22): given mem address in R0, print characters until NULL terminating character ('\0')

**GETC** (x20): takes character input from console and stores it in R0

---

# Complx

- An assembler and debugger for LC-3 assembly
- Available within the 2110 Docker container
- If you come to office hours asking for HW5/HW6 help, the first thing we'll ask is “have you tried debugging in Complx?”
  - Knowing how to do it yourself will save you a lot of time in the queue
  - Sets you up for success on quizzes/timed labs

---

# Loading

- Let's start with our OR.asm example from last time.
- To assemble a program and load it into the LC-3's memory, use File>Load in Complx.
- What do we notice?
  - What values are in R0-R7? The CC?
  - What values are in memory (besides our program)?
  - What value is in the PC?
  - You can customize the above using File > Advanced Load.
  - What happens to our pseudo-ops?
  - What happens to labels? (hint: View > Disassemble > Basic)

---

# Running

- It's as easy as clicking "Run"
- This will run through our program until it reaches HALT
- You can "Rewind" to start over
- We can also step through the program and watch the values of memory/registers change
- Note: by default, Complx labels and shows the instructions with labels instead of PC offsets, but there is actually no concept of labels at runtime

---

## Live coding with TRAPs

**Live coding example:** how can we print (using ASCII encoding) the sum of two numbers using PUTS?

**Note:** You can see any output from an assembly program in the floating Complx I/O window

---

## Debugging with Complx

- Let's load and run `arraysum.asm`.
  - Uses a while-loop as taught in lecture, how do these work?
- We know it assembles; there are no problems after File > Load.
- Unfortunately, the result isn't what we expect.
- Let's step through and try to find when things go wrong.

---

# Breakpoints and Watchpoints

- Sometimes, stepping through a whole program can be annoying.
  - What if your while-loop iterates 100 times, and the problem isn't until the 78<sup>th</sup> iteration?
- You can use breakpoints and watchpoints (Debug tab) to selectively stop your program
- Breakpoint: “stop when I reach this line”
- Watchpoint: “stop when a register or memory address value changes”
- You can also add a condition (e.g., only stop if R5-5 != 0) and “times” (e.g., only stop the first three times this occurs)

---

## Tips and Tricks

- Load replay strings using Test > Setup Replay String; this will let you step through your program with the exact inputs used for autograder test cases
- “View > Hide Addresses > Show only Code and Data” is sometimes helpful (it hides everything that isn’t part of your program)
- Use breakpoints and step through your program. Always ask “are these the register/memory values I expect?”

# Tips and Tricks

- Debug > Trace File creates a trace file that essentially steps though the whole program for you
- Check out “Canvas > Files > LC-3 Resources > ComplxMadeSimple.pdf” for a general reference!

```
PC x3000
instr: AND R0, R0, #0 (5020)
R0 24815|x60ef R1 27163|x6a1b R2 446|x01be R3 20906|x51aa
R4 -16914|xbdee R5 -22594|xa7be R6 32021|x7d15 R7 -13467|xcb65
CC: N
```

```
PC x3001
instr: AND R2, R2, #0 (54a0)
R0 0|x0000 R1 27163|x6a1b R2 446|x01be R3 20906|x51aa
R4 -16914|xbdee R5 -22594|xa7be R6 32021|x7d15 R7 -13467|xcb65
CC: Z
```

```
PC x3002
instr: LD R3, #12 (260c)
R0 0|x0000 R1 27163|x6a1b R2 0|x0000 R3 20906|x51aa
R4 -16914|xbdee R5 -22594|xa7be R6 32021|x7d15 R7 -13467|xcb65
CC: Z
```

```
PC x3003
instr: NOT R3, R3 (96ff)
R0 0|x0000 R1 27163|x6a1b R2 0|x0000 R3 5|x0005
R4 -16914|xbdee R5 -22594|xa7be R6 32021|x7d15 R7 -13467|xcb65
CC: P
```

---

# CS 2110 - Lab 09

## Subroutines & The Stack

Wednesday, October 13<sup>th</sup>, 2021



---

## Homework 5

- Basic assembly programming
- Released on Thursday, October 7<sup>th</sup>
- **Due Thursday, October 14<sup>th</sup> at 11:59 pm**
- Files will be available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until 10/15 applies

---

# Homework 6

- Covers assembly subroutines and calling convention
- Will be released on Thursday, October 14<sup>th</sup>
- **Due Wednesday, October 20<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)
- Standard grace period until Thursday 10/15
- **Will be demoed**

---

## Quiz 3

- Next Monday (October 18<sup>th</sup>)
- You must come to class and take the quiz here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TAs
- Full 75 minutes to complete the quiz!
- Topic list has been released on Canvas!

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 09
- 3) Access code: **stacksonstacks**
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



# Assembly Subroutines

- Subroutine is another name for function
- But there is no “function” abstraction in LC-3 assembly!
- So, we need to create our own abstraction

What do we need to have the function abstraction as in the C program on the right?

- `multiply()` should return to the correct place
- We need to communicate function parameters and return values somehow
- The state of other variables like `d` should not change after the call

```
#include <stdio.h>

int multiply(int op1, int op2) {
 int out = 0;
 for (int i = 0; i < op2; i++) {
 out += op1;
 }
 return out;
}

int main() {
 int a = 2;
 int b = 3;
 double d = 3.1415926;

 int c = multiply(a, b);
 printf("%d\n", a); // 2
 printf("%d\n", b); // 3
 printf("%d\n", c); // 6
 printf("%f\n", d); // 3.1415926
}
```

# JSR/JSRR and RET

|      |      |      |            |            |
|------|------|------|------------|------------|
| RET  | 1100 | 000  | <b>111</b> | 000000     |
| JMP  | 1100 | 000  | BaseR      | 000000     |
| JSR  | 0100 | 1    |            | PCoffset11 |
| JSRR | 0100 | 0 00 | BaseR      | 000000     |

- **JSR (Jump to Subroutine):** Saves the PC value into R7 and then sets PC to the target
    1.  $R7 = PC^*$
    2.  $PC = PC^* + PCOffset11$ 
      - PC-offset addressing (usually called with a label)
      - Note the order is important; we need to save the PC before changing it
  - **JSRR (Jump to Subroutine, Register):** Same as JSR, but uses a register for the subroutine's address instead of a label/offset
    1.  $R7 = PC^*$
    2.  $PC = SR$
  - **RET (Return):** A special case of JMP. Equivalent to JMP R7
    1.  $PC = R7$ 
      - This is actually a “pseudo-instruction” since it assembles to the exact same bits as JMP R7
- But is this enough to make subroutines?

---

# The Stack

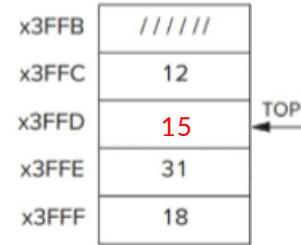
- The stack is a location in memory that is useful for storing temporary program data
  - Subroutine calls – parameters, return values
  - Local variables – what if our 8 registers aren't enough
- Grows “downwards” towards smaller memory addresses from a fixed starting location

# The Stack

- Grows “down” from some location in memory toward smaller addresses
- Top of stack is held in **R6**, aka the stack pointer



**Example:** pushing 15 (the value in R4) to the stack:



*What values are on the stack in each step?*

## Using the Stack

- Initialize R6 to some memory location  
(generally, the autograder will do this for you)
- PUSH data to top of the stack from some register RX

```
ADD R6, R6, #-1 ; move stack pointer
STR RX, R6, #0 ; store data
```

- POP data from top of stack into some register RX

```
LDR RX, R6, #0 ; restore data
ADD R6, R6, #1 ; move stack pointer
```

---

# Calling Subroutines

- The LC-3 calling convention!
- Create a **stack frame** (or **activation record**) on the stack that holds important information like parameters, return address, return value, etc.
- Save old register values in our stack frame so they can be restored

| TOP (LOWER MEMORY)     |  |                       |
|------------------------|--|-----------------------|
| param                  |  | <-R6 to call next sub |
| R4                     |  |                       |
| R3                     |  |                       |
| R2                     |  |                       |
| R1                     |  |                       |
| R0                     |  |                       |
| local                  |  | <-R5 (frame pointer)  |
| old FP (R5)            |  |                       |
| RA (old R7)            |  |                       |
| RV                     |  |                       |
| 1st param              |  |                       |
| xF000                  |  |                       |
| BOTTOM (HIGHER MEMORY) |  |                       |

---

# Frame Pointer

- The Frame Pointer (R5) holds the address of a fixed location in the stack frame
- Allows us to easily locate our local variables, arguments, return address, etc.
  - e.g., the return value is always R5+3
  - The first argument is always R5+4
- In the LC-3 calling convention, R5 points to the first local variable saved in our current stack frame

---

## Calling Convention: Important Registers

- R7 - current return address
- R6 - holds the stack pointer/top of the stack
- R5 - holds the current frame pointer

# Building up the stack

|           |                |
|-----------|----------------|
| Who Saves | STACK          |
| Callee    | Saved Regs     |
| Callee    | First local    |
| Callee    | oldFP/old R5   |
| Callee    | RA (callee r7) |
| Callee    | RV (space)     |
| Caller    | Arg 1          |
| Caller    | Arg 2          |

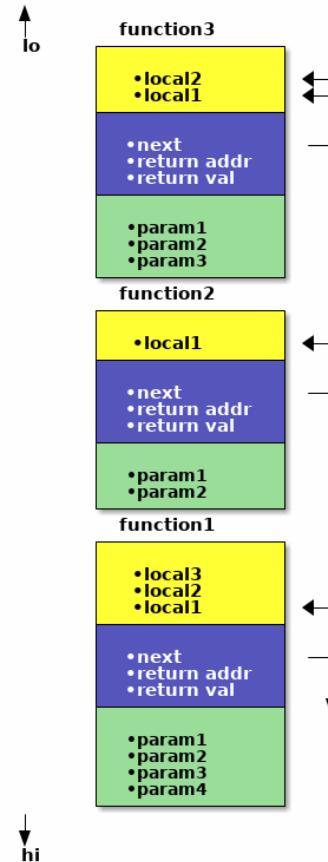
The diagram illustrates the stack layout. The stack grows from bottom to top. Arrows point from the 'Saved Regs' row to R5 and from the 'RA (callee r7)' row to R6.

Order of operations:

1. Caller pushes args in reverse order (argument 1 on top)
2. Caller uses JSR/JSRR to call subroutine
3. Callee allocates space for/saves: RV, RA, old FP (R5), at least one local variable
4. Callee allocates space for saving registers if need be
5. Reverse order at end of routine

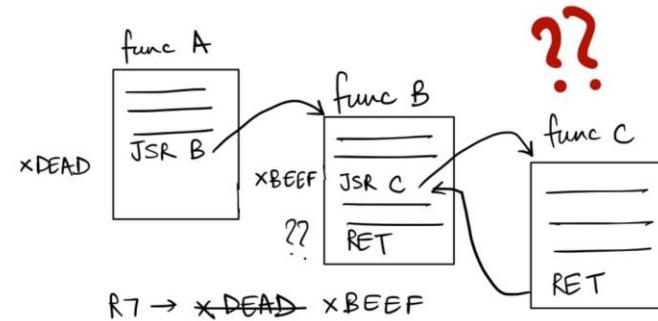
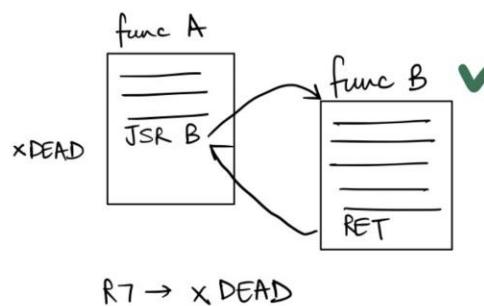
# What if a subroutine calls a subroutine?

- Just push another stack frame above it
- This works normally, and R6 will always point the current stack frame
  - A stack is a “last-in, first out” data structure
  - The most recently pushed stack frame will be that corresponding to the current function
  - Therefore, our current stack frame will always be at the top of the stack!
  - Ex: foo() calls bar(), which baz(). Since baz() was called most recently, it will be at the top of the stack.



# What if a subroutine calls a subroutine?

- However, there's a problem: R7 can get "clobbered" by the second call to JSR/JSRR, and then we won't know where to return to!



- This is why we save the return address (from R7) in our stack frame
- We simply have to restore R7 by popping the RA off the stack before calling RET

# Reviewing our abstraction

We said:

- `multiply()` should return to the correct place
  - We ensure this by saving R7, which is set by JSR/JSRR and returning to that location
- We need to communicate function parameters and return values somehow
  - Push these values to the stack
- The state of other variables like `d` should not change after the call
  - Save any registers we use and later restore them!

```
#include <stdio.h>

int multiply(int op1, int op2) {
 int out = 0;
 for (int i = 0; i < op2; i++) {
 out += op1;
 }
 return out;
}

int main() {
 int a = 2;
 int b = 3;
 double d = 3.1415926;

 int c = multiply(a, b);
 printf("%d\n", a); // 2
 printf("%d\n", b); // 3
 printf("%d\n", c); // 6
 printf("%f\n", d); // 3.1415926
}
```

---

## Some advice

- There is a lot of boilerplate involved in setting up and tearing down a stack frame, but it is almost the same for all functions
  - The only things that are different are the number of registers you save and the number of local variables
  - Make your life easier and use copy/paste for HW6!
- We don't grade on efficiency; there's no disadvantage to saving/restoring all registers R0-R4, even if you don't use some

---

## Live Coding Example

Write a program in assembly that recursively calculates:  
 $\text{sum}(n) = n + (n - 1) + (n - 2) + \dots + 3 + 2 + 1$

Pseudo code:

```
int sum (int n) {
 if (n <= 1) return 1;
 else return n+sum(n-1);
}
```

---

## Lab Assignment: Stack Subtract

- 1) Files on Canvas under: Files -> Lab Materials -> Lab 09
  - a) Extract the folder to the same directory as your cs2110docker.sh file.
  - b) Boot up Docker and complete the program.
- 2) Guide to using Complx Simulator:
  - 1) Files ->LC-3 Resources ->“ComplxMadeSimple.pdf

This is ungraded, but it is 100% in your best interest to do it before starting HW6. If you do it now, we can help you debug.

---

# CS 2110 – Labs 10+11

Lab 10 - Intro to C

Lab 11 – Storage, pointers,  
arrays and strings

Monday, October 25<sup>th</sup>, 2021



---

## Timed Lab 3

- This Wednesday (October 27<sup>th</sup>)
- You must come to class and take the TL here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TAs
- Full 75 minutes to complete the timed lab!
- Topic list shared by Shawn on Canvas (HW6-like problems)

---

## Homework 7

- Covers introductory C concepts
- Will be released on Thursday, October 28<sup>th</sup>
- **Due Wednesday, November 3<sup>rd</sup> at 11:59 PM**
- Files available on Canvas
- Submit to Gradescope (unlimited submissions)
- Standard grace period until Thursday 11/4

---

# Final Exam Logistics

- Two parts:
  - First part will be a Canvas Quiz
  - Second part will be Timed Lab-like assessment with Gradescope submission and local autograder
- Both parts will be open notes, open internet, but you must work alone and with just one device (the usual rules)
- Both parts will open and close at the same time; you must decide how to best budget your time
- More details will be released closer to the final exam

---

## Academic Honesty Policy (reminder)

- Academic misconduct is taken very seriously in this class
- It is in your best interest to do the work yourself
  - If you fall behind, turn in what you have and reach out to Caleb to get caught up
- Quizzes, timed labs, and the final are individual work
  - Sharing quiz codes is also an academic honesty violation
- You can only collaborate at a high level on homework
  - Any code must be your own
  - We use software to analyze homework submissions for cheating

---

## New TA Hiring

- Will be discussed in lab
  - Nov 1: B01, C01, C02, C03
  - Nov 3: B02, B03, B04, B05
- If you're interested in being a TA for CS 2110, this is for you!
- Many of the details also apply for being a TA for other DCI classes  
(basically any CS 1xxx or CS 2xxx)

---

## Lab Assignment: C Quiz!

- 1) Go to Quizzes on Canvas
- 2) Select Lab 11
  - Password: **pointer**
- 3) Get a 100% for attendance credit
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)

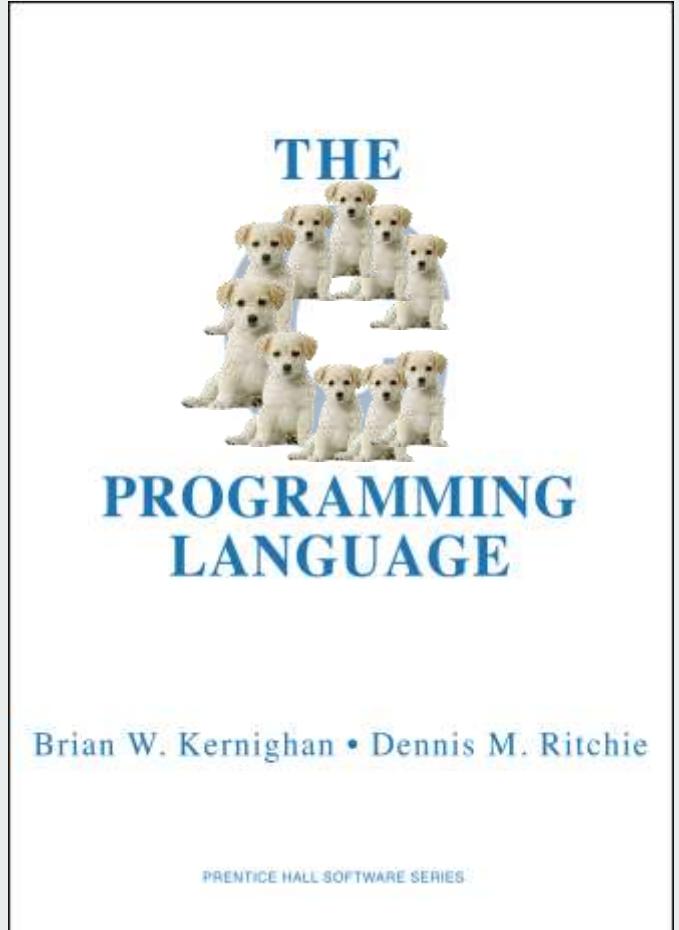
---

# CS 2110 - Lab 10

Intro to C

Wednesday, October 20<sup>th</sup>, 2021

[Click here to skip to lab 11 content!](#)



# C

- Developed in 1972 by Dennis Ritchie
- Somewhat like Java: compiled, statically-typed, has a similar syntax
- Manual memory management
- Low-level and procedural
  - Pretty close to assembly, but has convenient abstractions like functions
  - Not object-oriented; no classes

```
#include <stdio.h>
#include "main.h"

int y = 0;

/* this is a comment */
int main (void) {
 int x = 3;
 y = ADD(x, 2);
 printf("%d\n", y);

 return 0;
}
```



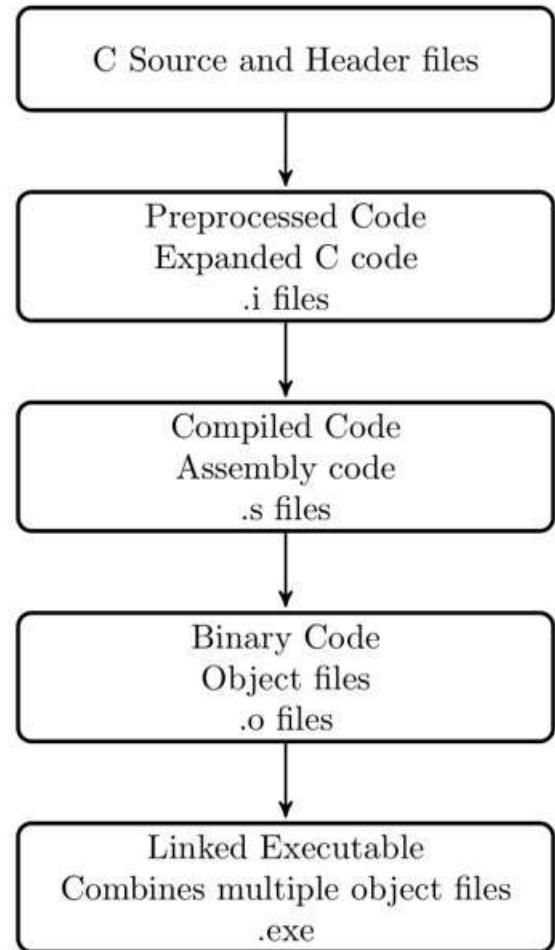
# C

- Used for systems programming, high-performance code, etc.
- It's more accurate to think about C as an abstraction on top of assembly, rather than "Java but different"
  - Remembering this will make C much easier to learn!
- Our textbook will be "The C Programming Language, 2<sup>nd</sup> edition" by K&R —available for free with your GT email!
  - See pinned Ed Discussion post #610

# Compilation Overview

---

- Step 1. Preprocessor
  - Runs before the compiler
  - We'll see what it can do in a bit!
- Step 2. Compiler
  - Convert C into assembly!
- Step 3. Assembler
  - You know this one already :)
- Step 4. Linker
  - Link multiple files, subroutines, etc. together





# Macros

```
#define MACRO_NAME(ARGUMENTS) TEXT_REPLACEMENT
```

Ex:

```
#define MULT(A,B) ((A)*(B))
#define PI 3.141593
```

If preprocessor sees MULT(5,7) somewhere in the C file it will replace it with ((5)\*(7))

PI will be replaced with 3.141593



# Macro Perils

What would the following program print?

Note: %d tells printf to print the parameter as a decimal number

```
#include <stdio.h>

#define MULT(a, b) a*b

int main(void) {
 printf("%d\n", MULT(2 + 3, 3));
}
```



# Macro Perils

Wrap every variable you use in parentheses:

```
#include <stdio.h>

#define MULT(a, b) (a)*(b)

int main(void) {
 printf("%d\n", MULT(2 + 3, 3));
}
```



# Macro Perils

What would the following program print?

```
#include <stdio.h>

#define SUM(a, b) (a)+(b)

int main(void) {
 printf("%d\n", 4 * SUM(2, 3));
}
```



# Macro Perils

Include outer parentheses when your macro produces an expression:

```
#include <stdio.h>

#define SUM(a, b) ((a)+(b))

int main(void) {
 printf("%d\n", 4 * SUM(2, 3));
}
```

---

# Header files

- In C, you can't use a function/variable before you declare it
- Header files contain function declarations and global variables
- They have a .h extension
- Like the "interface" of what a file exposes
- Shouldn't include function implementations

```
#ifndef MAIN_H
#define MAIN_H

#define ADD(x,y) ((x) + (y))

#endif
```

---

## #include ...

- C does not have an "import" system
- Preprocessor copies all C code from *filename* and replaces the include statement with that code
  - #include-ing a header file is like copy-pasting the declarations you need
  - Generally, you should never #include a .c file—why?
- #include <*filename*> for system header files
- #include "⟨*filename*⟩" for header files you write



# Include Guards

- What happens if I `#include` the same header file twice?
  - We'll get a compiler error due to multiple declarations
  - This can happen easily if a file includes two other files that share a dependency
- Solution: put this in our headers file to ensure each header file only gets included once:

```
#ifndef <HEADER_FILE_NAME>_H
```

```
#define <HEADER_FILE_NAME>_H
```

*Contents of header file*

```
#endif
```



# Include Guards Example

```
#ifndef MAIN_H /*if MAIN_H is already defined, skip to #endif*/
#define MAIN_H /*define MAIN_H*/

// Declares that a function int square(int x) is defined somewhere
int square(int x);

#endif
```

This code ensures that `square` is only declared once even if there are multiple `#include main.h` statements

---

## C Data Types and Sizes

- Unlike in Java, the exact sizes of number types is not specified in the C standard
- There are minimums in the standard as well as some conventions
- The exact sizes are implementation-specific and may vary between compilers or systems—you should never assume a given type is a given size
  - One exception: POSIX (another standard) requires `char` to be 8 bits, so we can pretty much always assume it is
- You can use the `sizeof` operator to determine the size of a C type (in bytes)

# C Data Types and Sizes

|             |                                         |
|-------------|-----------------------------------------|
| char        | Exactly 8 bits                          |
| short       | At least 16 bits (usually 16)           |
| int         | At least 16 bits                        |
| long        | At least 32 bits                        |
| long long   | At least 64 bits                        |
| float       | Unspecified, usually 32 bits            |
| double      | Unspecified, usually 64 bits            |
| long double | Unspecified, usually 64, 96 or 128 bits |

You can also safely assume the following about sizes:

`sizeof(long long) ≥ sizeof(long)`

`sizeof(long) ≥ sizeof(int)`

`sizeof(int) ≥ sizeof(short)`

`sizeof(short) ≥ sizeof(char)`

`sizeof(long double) ≥ sizeof(double)`

`sizeof(double) ≥ sizeof(float)`

`sizeof(float) ≥ sizeof(char)`

Pretty much, this means that no type is smaller than the one above it in the table (and float is no smaller than char).

---

# CS 2110 – Lab 11

Storage, pointers, arrays and strings

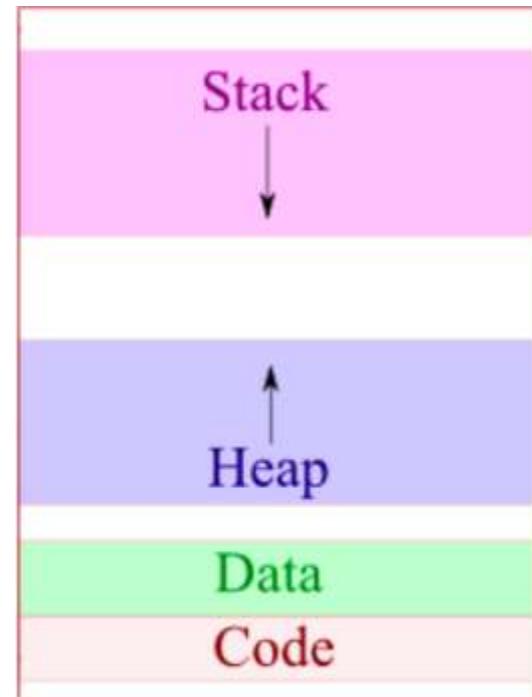
Monday, October 25<sup>th</sup>, 2021



Fig 1. The stack

# Memory Regions in C

- Just like in the LC-3, we have distinct memory regions
- Stack – local variables and arguments are stored here
  - Like the LC-3 stack!
- Heap – dynamically allocated memory
  - Related to allocating a “new” object in Java
  - We will learn more about this later
- Data – global variables are stored here
  - Like labels in your assembly program, e.g. ARRAY .fill x4000
- Code – our executable code



---

# Global Variables

x is a global variable stored in the "data" section of our memory

y is a local variable stored on the stack (specifically, on the stack frame for main)

```
int x = 4;
int main(void) {
 int y = 3;
 return 0;
}
```

---

## Type Qualifiers: `static`

- `static` defined *functions* are not visible outside of its C file (like `private` in java)
- `static` defined *global variables* are not visible outside of its C file (like `private` in java)
- `static` defined *local variables* do not lose values between function calls
  - Local variables are normally stored on the stack
  - This isn't possible for static locals, since they need to persist. Where can we store them instead?
- This distinction will be on a quiz.

```
foo(); /*x=1*/
foo(); /*x=2*/
foo(); /*x=3*/
foo(); /*x=4*/

void foo(void) {
 static int x = 0;
 x++;
 printf("x=%d\n", x);
}
```

---

## Type Qualifiers: `const` and `extern`

- `const` defines a variable as constant
- Ex: `const int x = 5;`
- `extern` tells the compiler that the variable has been defined in another file

*other.c*

```
int x = 5; /*global var*/
```

*main.c*

```
extern int x;
int main(void) {
 printf("%d", x) /*prints 5*/
}
```

# Pointers, Arrays, Strings & Pointer Arithmetic



---

# Pointers

- Pointers are variables that contain a *memory address*
- They also have a *type*
  - This refers to the type of the data AT that memory address.
  - There is also a special case: void pointers which point to a memory address but there is no type for the data at the address
- Denoted by an asterisk symbol following the type

---

# Pointer Examples

- Some pointer types:
  - `char *x;` // declares that x is a pointer to a char.
  - `char **y;` // declares that y is a *pointer to a pointer* to a char.
  - `void *z;` // declares that z is a pointer to an unspecified type
- An & symbol can be used to find the address of a code element such as a variable or function, and then assigned to a pointer:
  - `char i = 97;` // i stores the value 97 or 'a'
  - `char *x = &i;` // x stores the address of the variable i
  - `char **y = &x;` // y stores the address of the variable x
  - `void *z = &i;` // z also stores the address of the variable i

---

# Pointer Examples

```
int y = 7;
int z = 9;
int *x = &y;
x = &z;
```

One possible assembly implementation:

```
.orig x3000
LEA R0, Z ; R0 = x3022
ST R0, X ; x = x3022 = &z
.end
.orig x3020
X .fill x3021
Y .fill 7 ; address x3021
Z .fill 9 ; address x3022
.end
```

---

## Dereferencing Pointers

- ▷ We say that a pointer *points at* or *refers to* the memory value at the address contained in the pointer
- ▷ We can use the operator \* to *dereference* a pointer; in other words, to get the value that a pointer is pointing to.

- `char i = 97;` // i stores the value 97 or 'a'
- `char *x = &i;` // x stores the address of the variable I
  
- `x == 97;` // FALSE: the *address* stored in x is not 97
- `*x == 97;` // TRUE: the *value* at address x is 97
  
- `*x = 0;` // the value at address x is set to 0 or NULL (this changes i)
- `x = 0;` // x is now a NULL pointer (segfault if dereferenced)

# Dereferencing Pointers

```
int y = 7;
int *x = &y;
R0 = *x;
R0++;
*x = R0;
// C can't directly modify registers like
this, but this is C pseudocode ^_^(ツ)_/^
```

One possible assembly implementation:

```
.orig x3000
LDI R0, X ; R0 = mem[mem[x3021]]
ADD R0, R0, 1 ; R0++;
STI R0, X ; mem[mem[x3021]] = R0;
.end
.orig x3020
X .fill x3021
Y .fill 7 ; address x3021
.end
```

Now you see why LDI and STI might be useful!

---

# Arrays

- Arrays in C are much like arrays in assembly
- They declare a **fixed-size** sequence of same-typed elements.
- They are laid out consecutively in memory.
  - `int arr[5];` // declares arr as an array of 5 ints
  - `char arr[] = {'A',66,'C'};` // arr is an array of 3 chars
- You can also make arrays of pointers:
  - `int *arr[3];` // arr is an array of 3 *int pointers*,
  - 3 memory addresses where each memory address holds an `int`
- Unlike in Java, you cannot count on uninitialized data to be NULL or 0; it is simply *un-initialized* (just like assembly!).

---

# Arrays

- Like in assembly, arrays are essentially just two pieces of information: the address of the first element and the length of the array
  - `ARRAY .fill x4000`  
`LENGTH .fill 7`
- “Address of first element” == “pointer to first element”
- *Arrays decay to pointers*: we can choose to lose the length and implicitly convert an array to a pointer to its first element.
  - The following are equivalent:
    - `arr[0] = 7;`
    - `*arr = 7;`

---

## Sizes of C Types

- ▷ C has a special `sizeof(type_t)` operator that returns the size of a type.
  - What is the size of a type?
- ▷ Size of a type is expressed **in bytes**: i.e. a 32-bit integer has a size of 4 bytes.
- ▷ Remember, the size of a type is often **architecture dependent**.
  - Always use `sizeof()` to express/use the size of a type.
- ▷ Examples:
  - `sizeof(int)` = *depends (often 32 bits, 4 bytes)*
  - `sizeof(char)` = **1 byte**
  - `sizeof(long)` = *depends (often 64 bits, 8 bytes, 32 bits on Windows)*
- ▷ To avoid these variable sizes, there are explicitly-sized types in `<inttypes.h>`
  - `uint8_t`, `uint16_t`, `uint32_t` ...

---

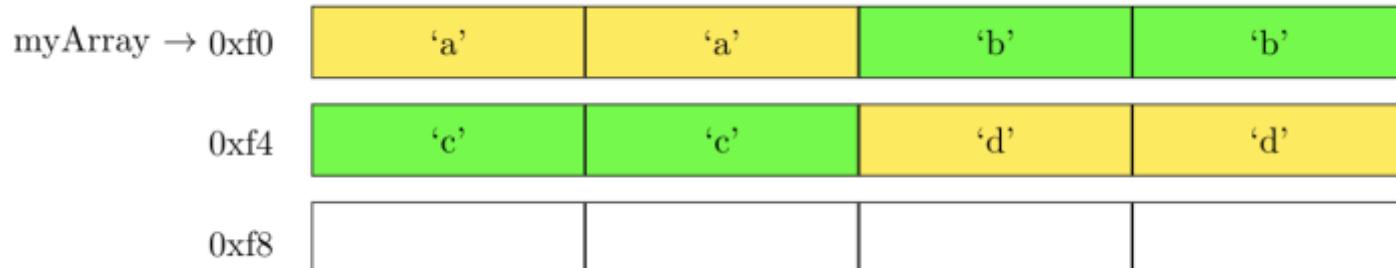
# Pointer Arithmetic

- ▶ **Pointer arithmetic** is the special way that C treats adding and subtracting to/from pointers; it adds **offset** times **the size of the pointer type**.
- ▶ So, using what we just learned, given:
  - `int *y;`
  - `y + 2` evaluates to `y + 2*sizeof(int)`
  - `y[2]` (array notation) is an equivalent shorthand
- ▶ This also applies to arrays:
  - `int arr[4];`
  - `arr[3] = 12;`
  - `*(arr + 3) = 12;`
- ▶ The above evaluates to `arr + 3*sizeof(int)` to get the correct physical address of the fourth element.

---

# Arrays

```
short myArray[] = {0x6161, 0x6262, 0x6363, 0x6464};
sizeof(short) == ? (answer given the layout below)
*(myArray + 2) == ?
```



---

# Strings

- ▷ Strings in C make use of all the concepts you just learned
- ▷ Strings in C are accessed through a pointer to the first character
  - `char *a;` //denotes a pointer to a character
- ▷ C-strings are **null terminated** (just like in assembly)
- ▷ You can think of this as an array of characters in memory.
- ▷ A string can be declared using either string literal or array notation:

```
char *a = "Hello"; // null terminator is added implicitly
char b[] = {'W', 'o', 'r', 'l', 'd', '\0'};
// you must add the null terminator
```
- ▷ Why is it important to have the null terminator?

---

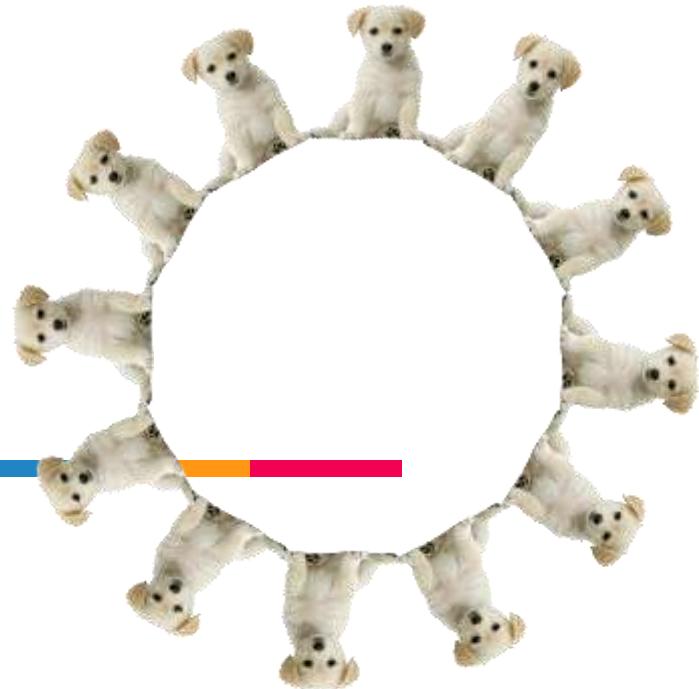
## A note on pointers and arrays

- ▶ Arrays can decay to pointers to their first element. And you can treat a pointer like an array, since array notation is a shorthand for pointer arithmetic.
  - `int *ptr;`
  - `int arr[10];`
  - `*(ptr + x) == ptr[x];`
  - `*(arr + x) == arr[x];`
- ▶ But they are not the same:
  - Arrays are fixed-size and cannot be reassigned. They refer to a specific region of memory.
  - A pointer is just a variable that holds a memory address.

# CS 2110 Lab 12:

## More C & Intro to GBA

### Monday, November 1<sup>st</sup>, 2021



# Homework 7 — Intro to C

- ▷ Covers introductory C concepts
- ▷ Released Saturday, October 28<sup>th</sup>
- ▷ **Due Wednesday, November 3<sup>rd</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Thursday 11/4

# Homework 8 — GBA

- ▷ Create an interactive graphical application for the GameBoy Advance using C!
- ▷ Released Thursday, November 4<sup>th</sup>
- ▷ **Due Wednesday, November 10<sup>th</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Thursday 11/11
- ▷ **Will be demoed**

# Lab Assignment: Canvas Quiz

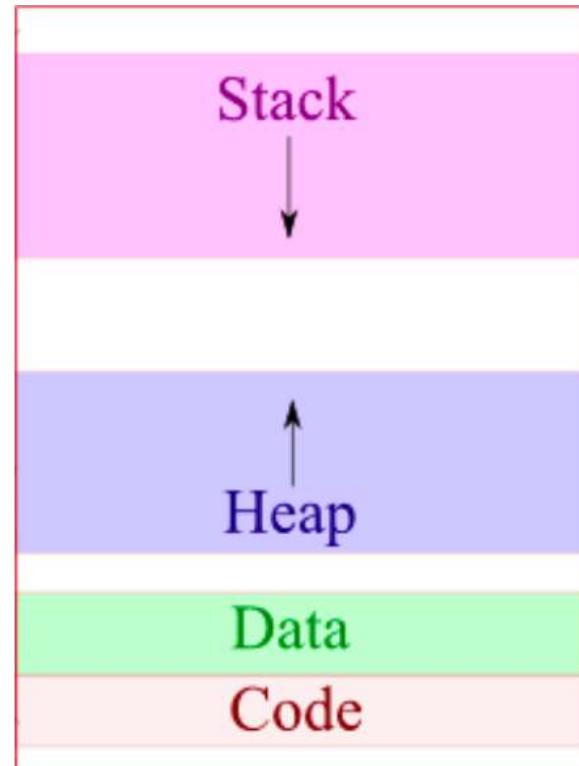
- 1) Go to Quizzes on Canvas
- 2) Select **Lab 12**
  - 1) Password: **0x2110**
- 3) Get 100% for lab attendance credit
  - a) Unlimited attempts
  - b) Collaboration is encouraged!
  - c) Ask your TAs for help :)

# More C!

(yep, there's more)

# Memory Layout

- ▷ Just like in the LC-3, there are multiple memory regions in a C program
- ▷ **Stack** – local variables and arguments are stored here
- ▷ **Heap** – dynamically allocated memory (covered later)
- ▷ **Data** – global and static variables are stored here
- ▷ **Code** – our executable code



# The Stack

- ▷ The stack in C will look very similar to the LC-3 stack
- ▷ We use it to store arguments, return values, local variables, etc.
- ▷ **This means this data is temporary.**  
When the function returns, the stack will be torn down, and the data on it becomes invalid garbage data.
- ▷ What is the output of the program to the right?

```
int main(void) {
 int *x = func();
 printf("%d\n", *x);
}

int *func(void) {
 int var = 42;
 return &var;
}
```

# Data & Code Segments

- ▷ Each function has its own stack frame, and it can't easily read the stack frames of other functions
- ▷ When we want global data, we need to store it somewhere that's accessible to all functions: the data segment
- ▷ Static local variables are also stored in the data segment as we don't want to lose their values when the stack frame is torn down
- ▷ Also, as we saw in the LC-3, we need somewhere in memory to put our code; this will go in the Code segment

# Structs

- ▷ In C, a struct (structure) is a data type that is used to group together multiple values
- ▷ They look similar to classes in Java; however, they do not have methods—they are just data bundled together
- ▷ Use the `.` operator to access variables inside a struct
- ▷ If given a pointer to a struct, use the `->` operator to simultaneously dereference and access it

```
struct Dog {
 char name[20];
 int isGood; // reminder: we use integers for Boolean values in C
};
```

# Structs in Memory

```
struct Dog {
 char name[20];
 int isGood;
 int age;
};
struct Dog my_dog;
```

- ▷ Assume `sizeof(int) == 4`
- ▷ What does `my_dog` look like in memory? Where is it located?

| Address       | At the Address     | Data Type |
|---------------|--------------------|-----------|
| x4000         | name[0]            | char      |
| x4001         | name[1]            | char      |
| x4002         | name[2]            | char      |
| x4003...x4013 | name[3] - name[19] | char      |
| x4014         | isGood             | int       |
| x4018         | age                | int       |

NOTE: Assume the struct starts at x4000

# C Pitfalls

```
struct Dog {
 char name[20];
 int isGood;
};
```

```
int main(void) {
 struct Dog dog = dogs[5];
 strncpy(dog.name, "Doggo", 6);
 dog.isGood = 1; // true
}
```

Let's say we have created a global array of dog structs

```
struct Dog dogs[10]; // stored in Data segment
```

Keep in mind that local variables are stored on the stack.  
Which `main` implementation will update the dog at index 5?

```
int main(void) {
 struct Dog *dog = &dogs[5];
 strncpy(dog->name, "Doggo", 6);
 dog->isGood = 1; // true
}
```

# C Pitfalls

- ▷ What is the output of the program to the right?
  - Hint: it's not what we want :(
- ▷ How can we adjust this program to get the expected output?
- ▷ When writing C, always be mindful of where your data is stored and how data is transferred between parts of your program

```
int main(void) {
 int x = 2109;
 func(x);
 printf("%d\n", x);
}
```

```
void func(int num) {
 num = num + 1;
}
```

# Graphics



# Graphics

- ▷ For HW8, you'll be using a Game Boy Advance emulator; to display the game, you need to learn about **GBA Graphics**.
- ▷ Pixel: 16-bit value



- ▷ Each color is allotted 5 bits; you can have an intensity from 0 – 31.
- ▷ Bit 15 is unused

# GBA Color Examples

Example colors:

- ▷ 0xFFFF → white
- ▷ 0x0000 → black
- ▷ 0x7C00 → pure blue
- ▷ 0x03E0 → pure green
- ▷ 0x001F → pure red
- ▷ 0x2110 → 

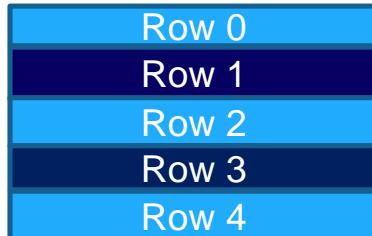


You will use a macro to simplify this conversion.

# GRAPHICS

- ▷ We use a “video buffer” to tell the GBA hardware what values we want to display on the screen
- ▷ The 'videoBuffer' is simply a **1-D array of pixels**.
  - But isn't our screen **2-D**?
  - Yes! The screen is:

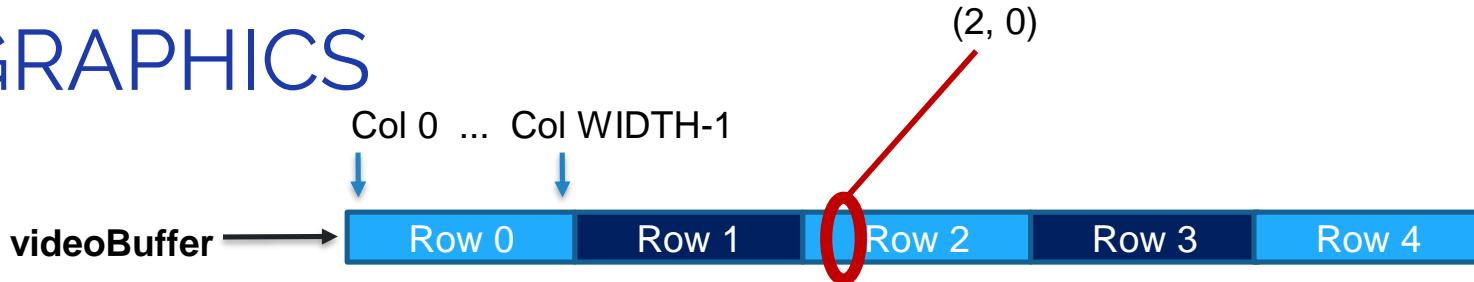
Displayed like this:



But **laid out** like this in memory:



# GRAPHICS



- ▷ Though our screen is stored like a 1-D array, we want a convenient way to index into a specific (row, col) coordinate.
- ▷ Offset calculation:

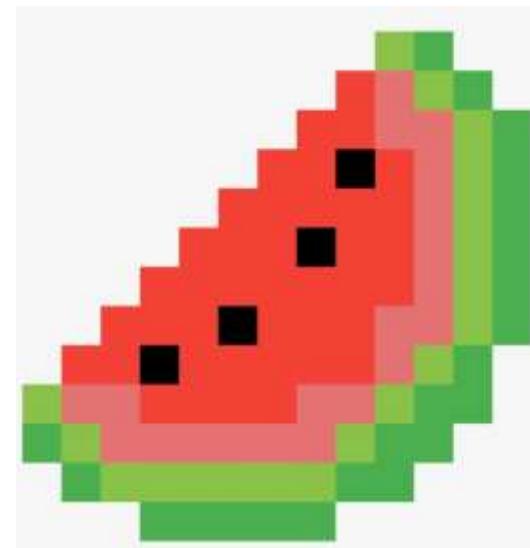
```
uint16_t videobuffer[];
videoBuffer(row, col) = videobuffer + (row * WIDTH + col);
```
- ▷ Note: WIDTH is the length of each row (the width of the screen)
- ▷ You will also use a macro to simplify this calculation.

# GRAPHICS

- ▶ What does this look like on a screen? (i.e.  $13 \times 13$  pixels)



`videoBuffer[6 * 13 + 7] = 0x0000;`



# GRAPHICS

- ▷ The GBA's screen dimensions:
  - **HEIGHT = 160**
  - **WIDTH = 240**
- ▷ How do we simulate movement?
  - We need to update the `videoBuffer` with the next position of our on-screen **moving** elements
  - We'll touch on this more in the future!

---

# CS 2110 Lab 13:

## More GBA & DMA

### Wednesday, November 3<sup>rd</sup>, 2021



# Homework 7 — Intro to C

- ▷ Covers introductory C concepts
- ▷ Released Saturday, October 28<sup>th</sup>
- ▷ **Due Wednesday, November 3<sup>rd</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Thursday 11/4

# Homework 8 — GBA

- ▷ Create an interactive graphical application for the GameBoy Advance using C!
- ▷ Released Thursday, November 4<sup>th</sup>
- ▷ **Due Wednesday, November 10<sup>th</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Thursday 11/11
- ▷ **Will be demoed**

# Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select **Lab 13**
  - 1) Password: **DMA**
- 3) Get 100% for lab attendance credit
  - a) Unlimited attempts
  - b) Collaboration is encouraged!
  - c) Ask your TAs for help :)

# GBA Review

- ▷ Each pixel is a 16-bit value, but bit 15 is unused
- ▷ Each color is allotted 5 bits; you can have an intensity from 0 – 31.

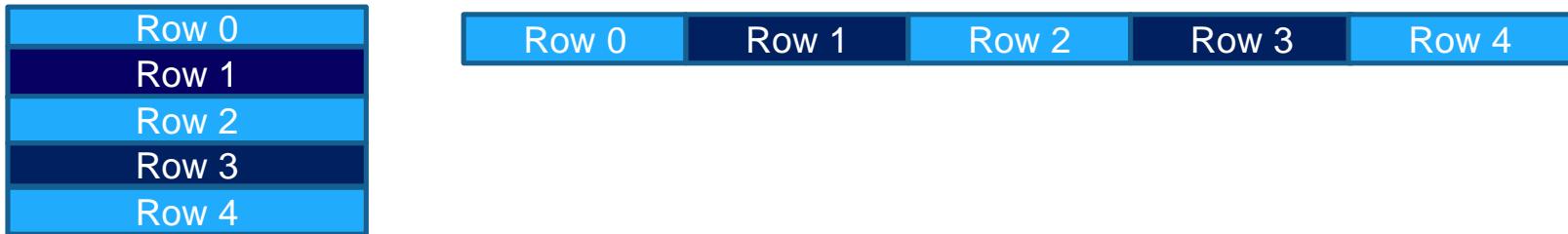


- ▷ We store the pixel values in a *memory-mapped array* called the *video buffer*
  - *Memory-mapped I/O*: when memory (the array) updates, the GBA hardware will update the corresponding hardware component (the screen)

# GBA Review

- ▶ `videoBuffer` is a one-dimensional array, but the screen is two-dimensional.

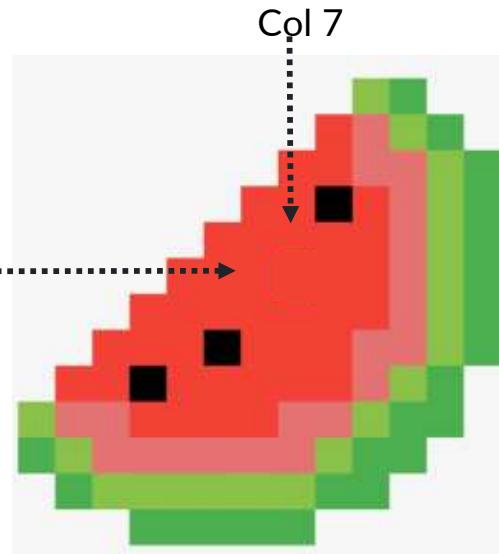
It is displayed like this, but laid out like this in memory.



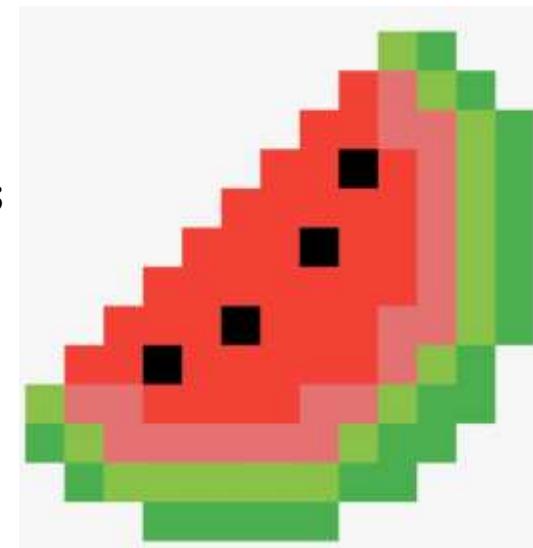
- ▶ Let `WIDTH` be the width of the screen (i.e., the length of each row)
- ▶ We can index into this array using the formula:  
 $\text{videoBuffer}[\text{row}][\text{col}] = (\text{row} * \text{WIDTH}) + \text{col}$

# GBA Review

- ▶ What does this look like on a screen? (i.e.  $13 \times 13$  pixels)



`videoBuffer[6 * 13 + 7] = 0x0000;`

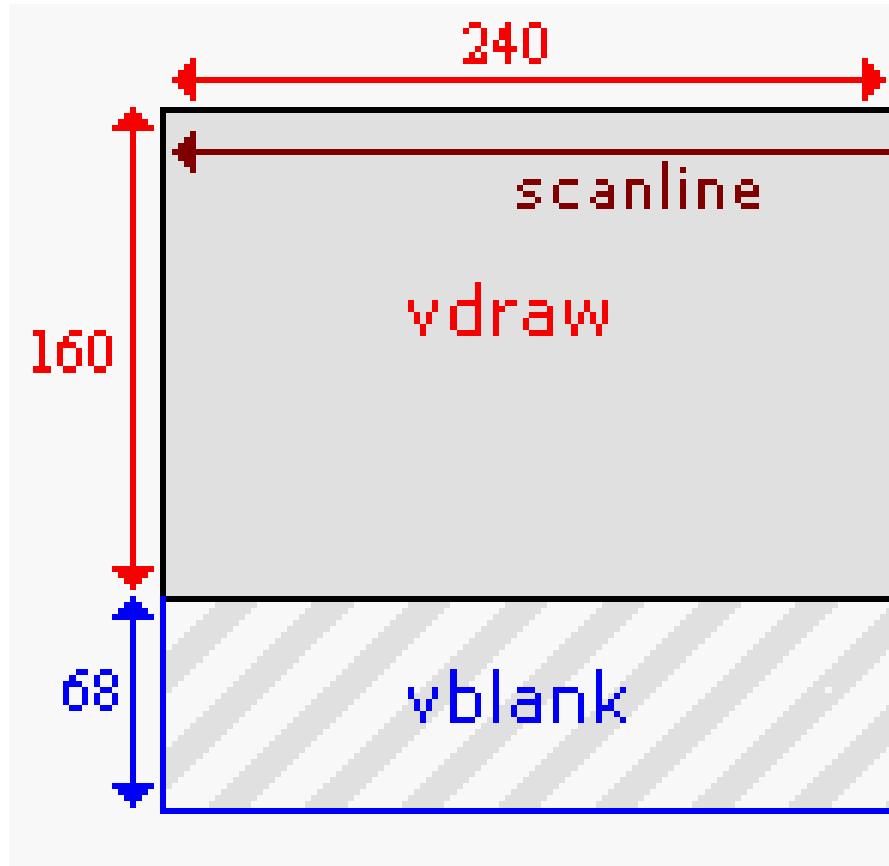


← →

**WIDTH = 13**

# GBA Draw Cycle

- ▶ The GBA alternates between two phases: **VDraw** and **VBlank**.
- ▶ **VDraw:** GBA copies one row of pixels at a time from the video buffer to the screen.
- ▶ **VDraw is not instantaneous:** halfway through VDraw, only half the scanlines have been drawn.
- ▶ **VBlank:** nothing happens.



# Tearing

- ▶ **Example:** Suppose all the pixels in the video buffer are red. We want to make the GBA screen blue.

Current video buffer:



What we want the screen to look like:



- ▶ Suppose VDraw starts while the video buffer is still all red, and then we update the video buffer to be all blue pixels before VDraw is completed. That would result in tearing (see the two examples below):

Resulting GBA Screen if the video buffer is updated early in VDraw:



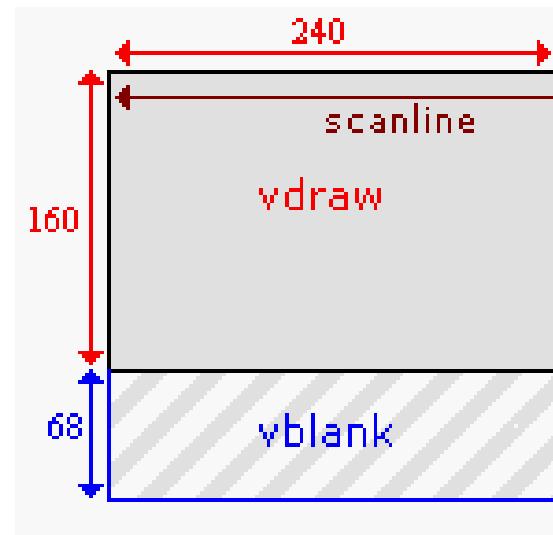
Resulting GBA Screen if the video buffer is updated later in VDraw:



- ▶ **Tearing:** the video buffer is updated during VDraw, causing the top half of the screen to show the old image and the bottom half to show the new image.
- ▶ If we update the video buffer during VBlank, no tearing will happen, because the screen is not updated during VBlank.

# Synchronizing Drawing and Logic

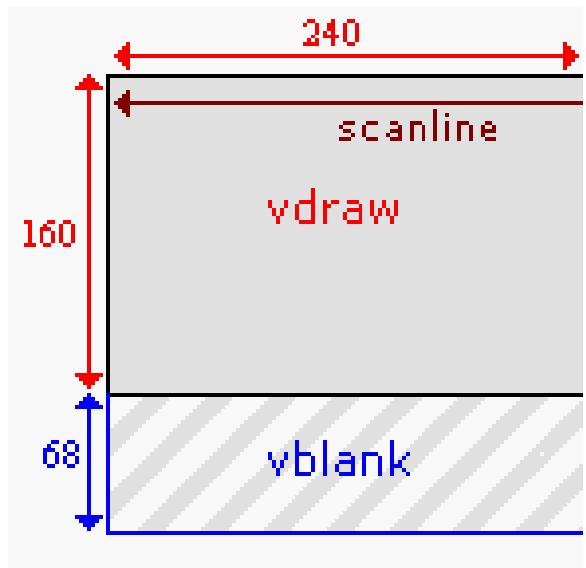
- ▷ Tearing looks bad, and we also need some way of synchronizing application logic.
  - Your HW8 applications are expected to have **no tearing!**
- ▷ The GBA exposes the *scanline counter* as a memory-mapped device register; it can be read using the macro defined below.
- ▷ The scanline counter indicates the current row of pixels being drawn at the screen (screen height: 160)
  - $0 \leq \text{SCANLINECOUNTER} \leq 159$ : VDraw
  - $160 \leq \text{SCANLINECOUNTER} \leq 227$ : VBlank



```
#define SCANLINECOUNTER *(volatile unsigned short *)0x40000006
```

# Implementing waitForVBlank

- ▷ How do we wait for VBlank?
- ▷ SCANLINECOUNTER > 160 → the GBA is in VBlank
- ▷ **Issue #1:** What if scanline is past 160, but almost at the **end of VBlank**? There's not enough time to change the video buffer before VDraw.
- ▷ **Issue #2:** What if app logic runs too quickly and we draw **two frames during the same VBlank**?
- ▷ To avoid these issues, we always wait for the **next full VBlank period**
  - SCANLINECOUNTER == 160 exactly
  - Wait until scanline is past VBlank and then comes back to VBlank

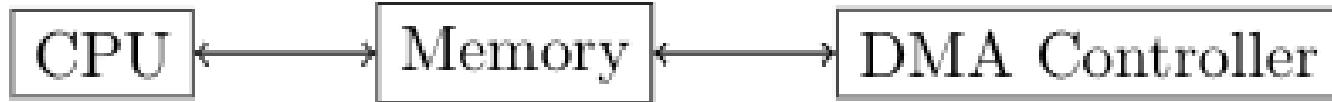


# Direct Memory Access (DMA)

- ▷ Drawing during VBlank helps prevent some tears, but unfortunately the GBA is not very powerful, and it's easy to draw too much and run out of VBlank time.
- ▷ Only about 1600 pixels updates can be drawn during a single VBlank phase. Filling the entire screen (38400 pixels) takes several cycles, guaranteeing tearing.
- ▷ Luckily, the GBA has a feature called Direct Memory Access, or DMA, which optimizes large array copies. Since filling the video buffer is basically an array copy, DMA is perfect for drawing!
- ▷ DMA is fast, but it isn't instant. It can't draw a full-screen image in a single VBlank cycle, but it can draw without tearing (before VDraw catches up).
- ▷ Like the scanline counter, DMA is manipulated using memory-mapped I/O.



# The DMA Controller



- ▷ The DMA controller is not connected to the CPU, but the CPU can control it by writing to specific memory locations.
- ▷ When DMA operates, it "steals" cycles from the CPU, and will return control to the CPU once it completes.
- ▷ There are 12 DMA registers for four DMA channels; each channel has a source, destination, and control register.
- ▷ We will use channel 3, the general-purpose channel.

# DMA Source and Destination Registers

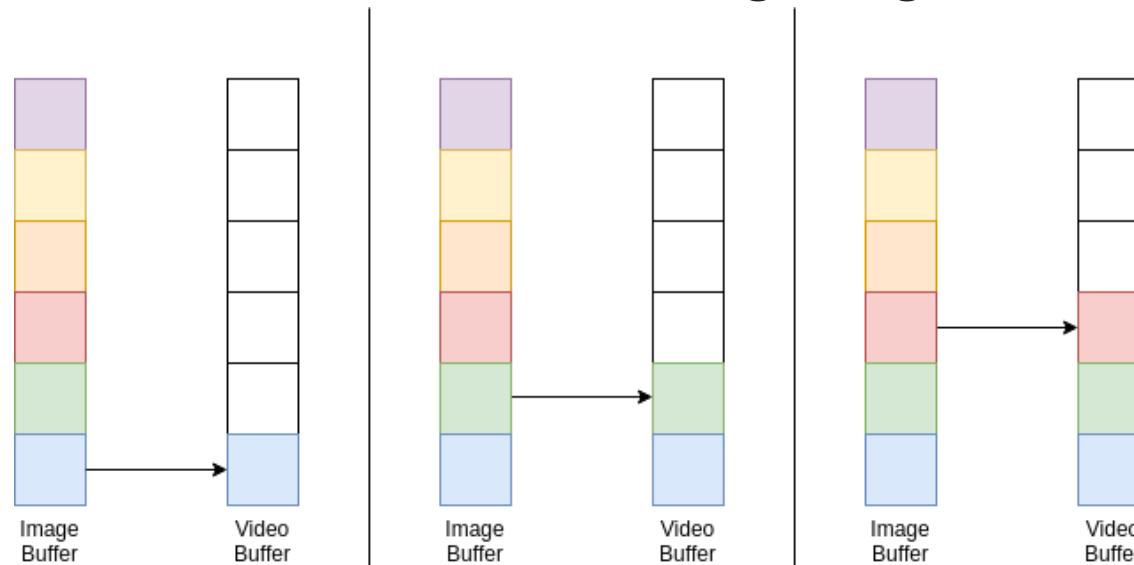
- ▷ Both the source and destination registers contain short pointers (addresses).
- ▷ The destination register will typically be given a pointer to `videoBuffer`, potentially with some offset added
- ▷ The source register will typically be given a pointer to an array of pixel values.
- ▷ Alternatively, the source register can be a pointer to a constant color.

# DMA Control Register

- ▷ The control register sets a few different things, using different bits:
    - Whether DMA is currently enabled (**En**)
    - How many elements to copy over (**N**)
    - How to iterate through the destination (**DA**)
    - How to iterate through the source (**SA**)
    - Whether to copy halfwords (16-bit) or words (32-bit) (**CS**)
  - ▷ Options for iterating through destination/source
    - Increment the address with each element (low → high addresses)
    - Decrement the address with each element (high → low addresses)
    - Fix the address with each element (don't change it)

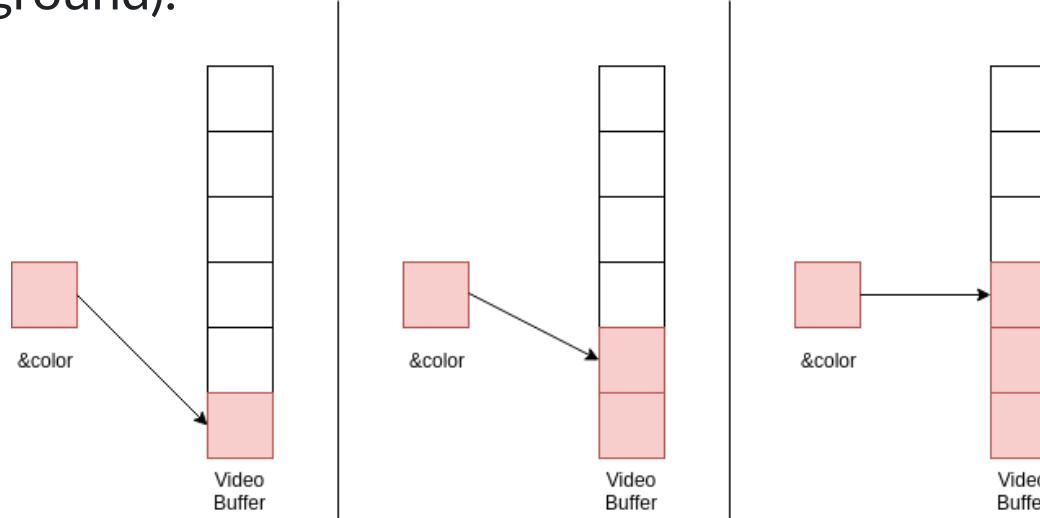
# DMA Example: Image on Screen

- When both the source and destination are set to increase after each copy, an element-wise copy will occur. This is useful for drawing images.



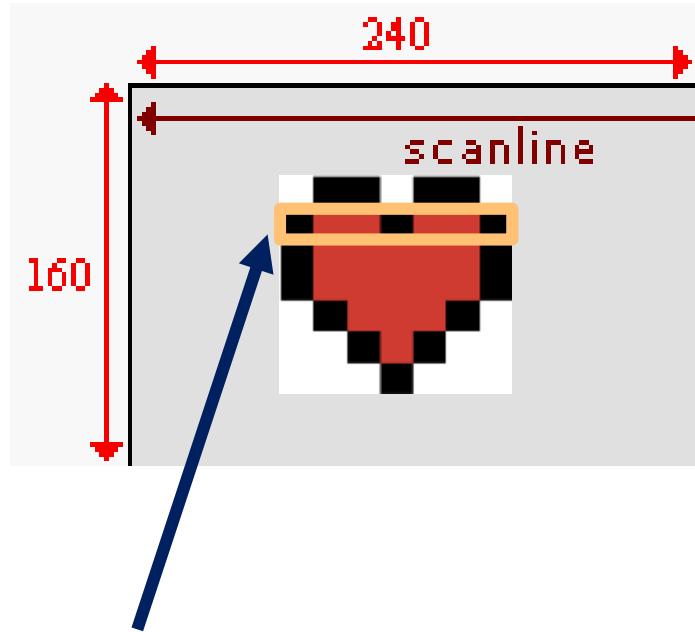
# DMA Example: Color on Screen

- ▷ If the destination is set to increment but the source is fixed, then a single value will be used to fill the entire buffer. This is useful for filling the screen with a single color (e.g., a solid background).



# Non-Full Screen Images

- ▷ So far, we have assumed a one-dimensional copy.
- ▷ Works well for the full screen, because each row of the image will be back-to-back without any other pixels in-between.
- ▷ However, DMA can only copy to and from contiguous addresses
- ▷ Consider that if we're copying a rectangle smaller than the full screen, we'll have to do the copy line-by-line.



Line will not be adjacent to the next line in the video buffer!

# Using DMA in code

- ▷ DMA\_CONTROLLER struct with the 3 registers:

```
typedef struct
{
 const volatile void *src;
 const volatile void *dst;
 u32 cnt;
} DMA_CONTROLLER;
```

- ▷ Array of 4 DMA controllers, one for each channel
  - DMA[3] is used to access channel 3
  - This is also memory-mapped I/O and allows us to control the DMA controller hardware

```
#define DMA ((volatile DMA_CONTROLLER *) 0x040000B0)
```

# GBA Program as a State Machine

- ▷ Typically, it is convenient to structure programs as a state machine, where each state draws different images and has different criteria for transitioning to other states.
- ▷ Example:
  - State 1: The start screen
    - Draws an image; transitions to State 2 when “start” is pressed
  - State 2: Core application
    - Contains the actual application—draws any necessary icons or images and transitions to State 3 if time expires
  - State 3: The “game over” or “application exited” screen
    - Draws an image; transitions to State 1 when “start” is pressed

# Buttons

- ▷ Another memory-mapped register the GBA exposes is the “key input” register
- ▷ In this register, bits 15–10 are unused, and each button is assigned a single bit. **The key states are low-active**, meaning their bit is cleared when the button is pressed and set when it is released.
- ▷ How can we extract a single bit from this whole number?
  - Hint: think back to HW1
- ▷ Example: how can we get the value of the start button?
  - `#define BUTTON_START (1<<3)`

|   |   |   |      |    |      |       |       |        |   |   |   |   |   |   |   |
|---|---|---|------|----|------|-------|-------|--------|---|---|---|---|---|---|---|
| F | E | D | C    | B  | A    | 9     | 8     | 7      | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| - | L | R | down | up | left | right | start | select | B | A |   |   |   |   |   |

```
#define BUTTONS *(volatile u32 *) 0x4000130
```

# Changing States on Button Presses

**Idea:**

```
#define KEY_DOWN(key, buttons) (~buttons) & (key)
if (KEY_DOWN(BUTTON_A, buttons)) {
 state = next_state;
}
```

**Problem:**

- ▷ What if our next state wants to transition by pressing A also?
- ▷ Most programs will run at 60 frames per second, so if the above code runs on the next frame, the player will only have a sixtieth of a second to lift the A button if they don't want to change to the next state.

# Edge-Triggered State Changes

**Solution:** keep track of the previous button state

```
previousB = currentB;
currentB = buttons;
if (KEY_DOWN(BUTTON_A, currentB) &&
 !KEY_DOWN(BUTTON_A, previousB)) {
 state = next_state;
}
```

- ▷ The above code will change the state only if the A button is currently pressed but was not pressed the last time the buttons were checked.
- ▷ You will have to write a KEY\_JUST\_PRESSED macro to make the above more concise.

# Good Resource For Learning GBA!

- ▷ <https://www.coranac.com/tonc/text/>
- ▷ Covers all the topics discussed here, and much much more!
  - Sprites
  - Tilemaps
- ▷ Note that if you try to implement audio (not taught in lecture/lab), the Docker environment will probably not support it

# Lab Assignment: Intro to DMA

- 1) Go to Lab Assignments → lab13.zip on Canvas
- 2) Extract the folder to the same directory where `cs2110docker.sh` is located (or a subdirectory of it)
- 3) Boot up Docker and open **README.txt** to get started!
- 4) When you're done, complete the lab quiz (Canvas → Quizzes → Lab 13) for attendance credit!
  - 1) Quiz code: **DMA**
  - 2) You still must complete the quiz for attendance credit!



---

# CS 2110 Lab 14:

## Debugging with GDB

### Monday, November 8<sup>th</sup>, 2021

# Homework 8 — GBA

- ▷ Create an interactive graphical application for the GameBoy Advance using C!
- ▷ Released Thursday, November 4<sup>th</sup>
- ▷ **Due Wednesday, November 10<sup>th</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Thursday 11/11
- ▷ **Will be demoed**

# Quiz 4

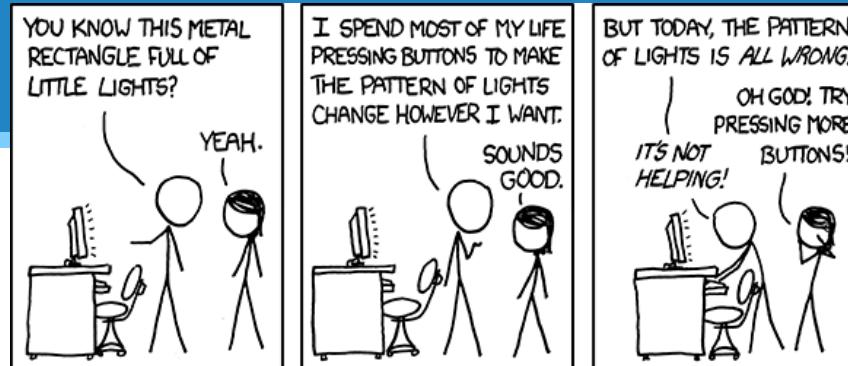
- ▶ Next Wednesday (November 17<sup>th</sup>)
- ▶ You must come to class and take the quiz here
- ▶ Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TA(s)
- ▶ Full 75 minutes to complete the quiz!
- ▶ Topic list will be released on Canvas
  - Will cover mostly C topics

# Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select **Lab 14**
  - 1) Password: **backtrace**
- 3) Get 100% for lab attendance credit
  - a) Unlimited attempts
  - b) Collaboration is encouraged!
  - c) Ask your TAs for help :)

# GDB

*debugging, made better*



# Debugging

- ▶ A lot of development time is spent fixing code
- ▶ Logical errors and runtime errors are generally the harder ones to fix
- ▶ It is possible to use `printf` for basic debugging
- ▶ However, using tools specialized for the job is far more productive, particularly for more complex bugs and software
  - We've seen this earlier in the semester with Complx

# GDB

- ▷ gdb → GNU Debugger
- ▷ Command line tool that lets you inspect and debug your code from within a shell/terminal
- ▷ Very powerful and can save you HOURS of time
- ▷ For this class, it is much easier if you just run it from within Docker
  - It can also be installed locally using your package manager of choice (brew, apt-get, nix, dnf, ...)
  - Will not work on M1 Macs! We are actively working on a fix/workaround

# Getting started with GDB

- ▶ First, we need to compile our code for gdb
  - `gcc -g -o <name-of-output-file> <source-file>`
- ▶ `-g` tells the compiler to add debug information, which GDB needs in order to work
- ▶ `-o` lets you rename your output file (the default filename is `a.out`)
- ▶ The Makefile we give you does this for you automagically when you run `make tests`
  - Unfortunately, Mednafen doesn't support GDB so this will not work for HW8

# GDB Commands

- ▶ `gdb <name-of-output-file>`
  - Starts gdb and loads it with <name-of-output-file>
- ▶ Alternatively, with our Makefile:
  - `make run-gdb TEST=<name of test case>`
  - `make run-gdb TEST=test_catchPokemon_single`

The following commands must be run from within gdb:

- ▶ `run <args>`
  - Runs the program to be debugged
  - Whatever arguments you pass in will be passed along to the program

# GDB Commands (continued)

- ▷ `break <where>`
  - Sets a new breakpoint at `<where>`
  - `<where>` = function-name | line-number | file:line-number
- ▷ `watch <what>`
  - Sets a new watchpoint on a variable or expression
  - The program will pause when `<what>` changes
- ▷ `step`
  - Run the next line of code, stepping into any functions
- ▷ `next`
  - Run the next line of code, stepping over any functions

# GDB Commands (continued)

- ▷ `print <what>`
  - Prints `<what>` to the console
  - `<what>` = variable-name | expression
- ▷ `display <what>`
  - Prints `<what>` to the console, but every step after it will continue to print `<what>`
- ▷ `backtrace`
  - Displays the call stack
  - Useful for debugging seg faults

# GDB Commands (continued)

- ▶ `finish`
  - Finish the current function execution (i.e. run every line until you return to the caller)
- ▶ `record`
  - Record history starting now
  - You can go back using this history using the `reverse-step` or `reverse-next` commands
  - You cannot go backwards unless you record history

# Debugging Example

- ▶ Your catchPokemon function is returning FAILURE instead of SUCCESS for the “multiple\_1” test case
- ▶ How do we open our code in GDB?
  - make run-gdb TEST=test\_catchPokemon\_multiple\_1
- ▶ How can we stop when we reach our function?
  - break catchPokemon
  - run
- ▶ Let’s go to the end of the function so we can see which return statement is triggered, print values, etc. We could step through, but is there a faster way?
  - record
  - finish
  - reverse-step

# Debugging Example

- ▷ The value for `pokedex[2].pokedexNumber` isn't getting set correctly, but you don't know where it's getting set
- ▷ One option would be to display the value, and then step through until it changes
  - `display pokedex[2].pokedexNumber`
  - `step/next`
- ▷ Even easier would be to set a watchpoint on the value, and then run and it'll pause when the value is updated!
  - `watch pokedex[2].pokedexNumber`
  - `run`
  - `continue`

# Live Debugging

▷ string\_indexof.c

```
malloc(3 * 5 * sizeof(struct lab))
```



---

# CS 2110 Lab 15:

## Type Declarations and Dynamic Memory

### Wednesday, November 10<sup>th</sup>, 2021

# Homework 8 — GBA

- ▶ Create an interactive graphical application for the GameBoy Advance using C!
- ▶ Released Thursday, November 4<sup>th</sup>
- ▶ **Due Wednesday, November 10<sup>th</sup> at 11:59 PM**
- ▶ Files available on Canvas
- ▶ Submit on Gradescope (unlimited submissions)
- ▶ Standard grace period until Thursday 11/11
- ▶ **Email Shawn by tonight if you don't have a demo slot to be guaranteed one**

# Homework 9 – Linked List

- ▷ Apply dynamic memory knowledge (today's lab!) in C
- ▷ Will be released Thursday, November 18<sup>th</sup>
- ▷ **Due Tuesday, November 30<sup>th</sup> at 11:59 PM**
- ▷ Files available on Canvas
- ▷ Submit on Gradescope (unlimited submissions)
- ▷ Standard grace period until Wednesday 12/1

# Quiz 4

- ▶ Next Wednesday (November 17<sup>th</sup>)
- ▶ You must come to class and take the quiz here
- ▶ Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TA(s)
- ▶ Full 75 minutes to complete the quiz!
- ▶ Topic list will be released on Canvas
  - Will cover mostly C topics

# Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select **Lab 15**
  - 1) Password: **ramimalloc**
- 3) Get 100% for lab attendance credit
  - a) Unlimited attempts
  - b) Collaboration is encouraged!
  - c) Ask your TAs for help :)

# Reading C Type Declarations

```
char *(*(**foo[][8])(int))[]; // huh?????
```

# Review

Describe these types in English:

- ▷ `char *c;`
- ▷ `int arr[8];`
- ▷ `char *argv[];`
- ▷ `int (*matrix)[8];`
- ▷ `int (*add)(int, int);`

# Derived Types

All types have:

- ▷ exactly one **base type**: int, char, struct coord, etc.
- ▷ zero or more **derived types**:
  - \* — "pointer to..."
  - [] — "array of..."
  - [n] — "array of n ..."
  - ... (...) (args) — "function taking args and returning ..."

# How do I read a declaration?

1. Find the identifier name, e.g. "x" in "int x;"
2. Read as far right as you can until:
  - You hit a close paren ')', or
  - You reach the end of the declaration
3. Go back and read as far left as you can until:
  - You hit an open paren '(', or
  - You reach the beginning of the declaration
4. If you hit parentheses, exit the parentheses
5. Go to 2 and repeat until you read the whole declaration

# Example

```
long ** foo[7];
```

## Example – find identifier name

```
long ** foo[7];
```

"foo is..."

# Example – read right as far as you can

```
long ** foo[7];
```

"foo is an array of 7..."

We hit the end of the declaration, so we go back and read left

## Example – go back and read left

```
long ** foo[7];
```

"foo is an array of 7 pointers to..."

```
long ** foo[7];
```

"foo is an array of 7 pointers to pointers to..."

```
long ** foo[7];
```

"foo is an array of 7 pointers to pointers to longs"

# Another Example

```
int (*foo[5])(char*);
```

"foo is..."

Where do we go next?

# Another Example

```
int *(*foo[5])(char*);
```

"foo is an array of 5..."

We hit a ')', so we go back and read left

## Another Example

```
int (*foo[5])(char*);
```

"foo is an array of 5 pointers to..."

We hit a '(', so we exit the parens and read right again

## Another Example

```
int *(*foo[5])(char*);
```

"foo is an array of 5 pointers to functions taking a  
char\* and returning..."

We hit a '(', so we exit parens and read right again

## Another Example

```
int *(*foo[5])(char*);
```

"foo is an array of 5 pointers to functions taking a  
char\* and returning a pointer to..."

```
int *(*foo[5])(char*);
```

"foo is an array of 5 pointers to functions taking a  
char\* and returning a pointer to int"

# Optional Challenge

```
char *(*(**foo[][8])(int))[];
```

(We won't ask you anything this hairy!)

# Good Resource: cdecl

- ▷ <https://cdecl.org/>
- ▷ Try playing around with different type declarations!

cdecl

C gibberish ↔ English

```
long ** foo[7];
```

declare foo as array 7 of pointer to pointer to long

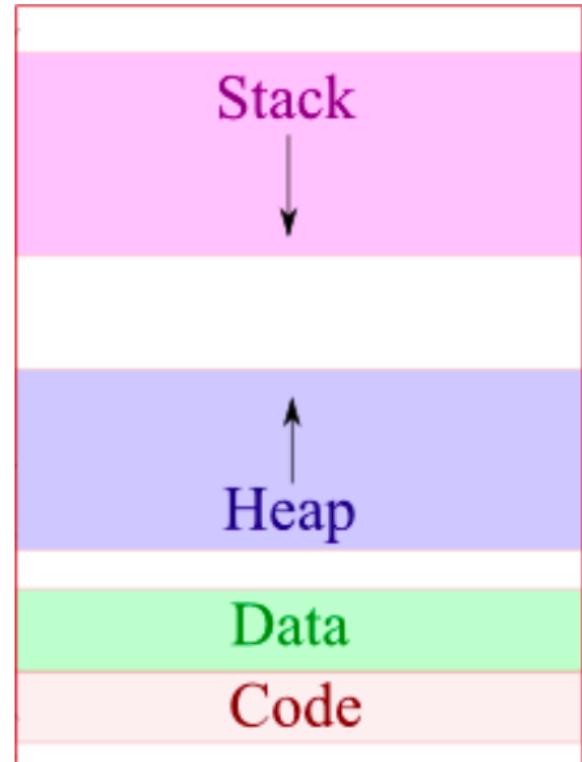
[permalink](#)

# Dynamic Memory Allocation

Fun with `malloc` and friends!

# Memory Layout

- ▷ Let's revisit this diagram one more time:
- ▷ **Stack** – local variables and arguments are stored here
  - What are the disadvantages of stack allocation?
- ▷ **Data** – global and static variables are stored here
  - What are the disadvantages of the data segment?
- ▷ **Code** – our executable code



What if we want memory that is both *dynamically allocated* and *persistent*?

# The Heap!

- ▷ Sometimes, we write programs expecting user input or other input from a source we don't control (such as a file)
- ▷ Accordingly, we may not know the exact number of variables or structs we will need.
- ▷ In these cases, we utilize the heap.
- ▷ The heap holds dynamically allocated variables. These are variables whose space is allocated at runtime.
  - Like creating a new object in Java

# Malloc

- ▶ If we want to dynamically allocate memory, we can use the `malloc()` function
  - ▶ Malloc takes in one argument: how many bytes to allocate from the heap
  - ▶ It returns a pointer to the allocated memory, or `NULL` if there's an error
  - ▶ Example:
    - `int *myInt = malloc(sizeof(int));`
    - Where is `myInt` stored? Where is `*myInt` stored?
- myInt now points to  
a block of four bytes  
on the heap, assuming  
`sizeof(int) == 4`

# Free

- ▶ C does not have garbage collection like Java; when we are done with memory, we need to give it back to the heap so that it can be allocated for later use
- ▶ `free()` tells C that it can give the memory to other code, which will use it for other purposes
- ▶ **Never use a freed pointer, as this leads to undefined behavior, only free() a block when you're done with it**
- ▶ What if we forget to `free()` a block that we allocate?
- ▶ Example: `free(myInt)`

# Calloc

- ▶ While `malloc()` allocates a memory block of given size and returns a pointer to the beginning of the block, `malloc()` doesn't initialize the allocated memory—it is garbage data.
- ▶ If needed, we can use the `calloc` function to avoid this which will allocate the heap space and initialize it to zero.
- ▶ Unlike `malloc`, `calloc` takes 2 arguments: the number of elements and the size of each element (useful for arrays!)
- ▶ Example:
  - `int *array = calloc(array_len, sizeof(int))`

# Realloc

- ▷ Sometimes, we may wish to grow or shrink the size of a block on the heap. This may or may not actually use the same physical memory block on the heap, but, regardless, we want to preserve the data present
- ▷ In these cases, we use the `realloc` function
- ▷ Example:
  - `int *first5 = realloc(numbers, 5*sizeof(int));`
  - `first5` is a pointer to the first 5 ints of the “array” `numbers`
  - Note that the pointer `numbers` is no longer valid

# Realloc

- ▶ Let's say we have a heap-allocated `int` array of size 30, and we want to double its size to 60.
- ▶ What if we tried:
  - `array = realloc(array, sizeof(int) * 60);`
- ▶ This seems to work, and it does in most cases!
- ▶ What could go wrong, and how do we fix it?

# Uses of Malloc

- ▶ malloc (and friends) are our way of interacting with the heap and gaining access to memory that isn't on the stack.
- ▶ Some possible uses of malloc:
  - Dynamically allocated arrays (1D or multi-dimensional)
  - Dynamic data structures like linked lists
  - Large data structures that won't fit on the stack
  - When the data needs to survive after the function ends
- ▶ Stack allocation should be preferred when possible as it is faster

# Dynamically Allocated 2D arrays

- ▶ Let's say we wish to make a 2D array of ints on the stack.
  - We can do this very easily: `int grid[3][3]`
- ▶ We can also dynamically do this on the heap, like so:
  - `int *grid2 = (int*) malloc(3 * 3 * sizeof(int))`
- ▶ This will create a very similar array to the one above; however, we can't index it using the standard notation: `grid2[row][col]`
  - This is because the compiler doesn't know that the allocated space is being used as a 2D array
- ▶ How can we access values in `grid2`?
  - Hint: think about the GBA `videoBuffer`

---

# CS 2110 - Lab 16

Malloc Usage, Error Handling  
and Unions

Monday, November 15<sup>th</sup>, 2021



---

## Homework 9 – Linked List

- Apply dynamic memory knowledge in C
- Will release Thursday, November 18<sup>th</sup>
- **Due Tuesday, November 30<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until Wednesday 12/1
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

## Quiz 4

- This Wednesday (November 17<sup>th</sup>) during lab
- You must come to class and take the quiz here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TA(s)
- Full 75 minutes to complete the quiz!
- Topic list released on Canvas
  - Will cover mostly C topics

---

# TA Applications are open!

- If you're interested in being a TA for 2110 or any other lower-division course in the CoC, applications are open!
- <http://ta-app.cc.gatech.edu/>
- If you're not using campus Internet, then you'll need the VPN to access the page
- **Applications close on Monday, November 22**

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 14
- 3) Access code: **union ring**
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



---

# Dynamic Memory Allocation — Review

---

## Dynamic Memory Allocation

- When we allocate data on the stack, it is short-lived
- When we allocate data in the data segment, we need to know how much to allocate **at compile-time**
- To dynamically allocate persistent data, we need to allocate space on the *heap*

---

## Malloc & Friends

- `malloc(size)`: allocates `size` bytes on the heap
- `calloc(nmemb, size)`: allocates `nmemb*size` bytes on the heap and zeroes them out
- `realloc(ptr, size)`: the “resize” function; returns a block of `size` bytes with the data from `ptr`
- These functions return a `void*` to the new block, or `NULL` on failure

---

## Free

- `free(ptr)`: Gives a pointer returned by `malloc` (or friends) back to the heap; we no longer need it
- **Never use a freed pointer, it is no longer valid.**
- Failing to free pointers you're not using will cause memory leaks; the heap will run out of space
- `realloc(ptr, size)` will also free `ptr`, so you should use the pointer returned instead

---

# Malloc Error Handling

```
int *get_array(void) {
 int *ptr = calloc(4, sizeof(int));
 ptr[2] = 7; // can this segfault?
 return ptr;
}
```



# Malloc Error Handling

```
int *get_array(void) {
 int *ptr = calloc(4, sizeof(int));
 if (!ptr) return NULL;
 ptr[2] = 7;
 return ptr;
}
```

**Solution:** always check the result of malloc & friends! If it returns NULL, handle it appropriately

---

# Malloc Error Handling

```
int *get_array(void) {
 int *ptr = malloc(4*sizeof(int));
 if (!ptr) return NULL;
 ptr[2] = 7;
 ptr = realloc(ptr, 6*sizeof(int));
 if (!ptr) return NULL;
 ptr[5] = 12;
 return ptr; // is there a problem now?
}
```

---

## Malloc Error Handling

```
int *get_array(void) {
 int *ptr = malloc(4*sizeof(int));
 if (!ptr) return NULL;
 ptr[2] = 7;
 int *new_ptr = realloc(ptr, 6*sizeof(int));
 if (!new_ptr) { free(ptr); return NULL; }
 ptr = new_ptr;
 ptr[5] = 12;
 return ptr;
} Solution: always assign the result of realloc to a new pointer!
```



## Warning:

```
struct dinosaur *create_dino(char *name) {
 struct dinosaur *dino = malloc(sizeof(struct dino));
 if (!dino) return NULL;
 dino->name = name;
 dino->coolness = 11; // Example:
 return dino;
}
```

```
struct dinosaur {
 char *name;
 int coolness; // scale from 1-10
}
```

What happens if name points to the stack? Or even the data segment?



```
struct dinosaur {
 char *name;
 int coolness; // scale from 1-10
}
```

## Solution:

```
struct dinosaur *create_dino(char *name) {
 struct dinosaur *dino = malloc(sizeof(struct dino));
 if (!dino) return NULL;
 dino->name = malloc(strlen(name) + 1);
 if (!dino->name) { free(dino); return NULL; }
 strcpy(dino->name, name);
 dino->coolness = 11;
 return dino;
}
```

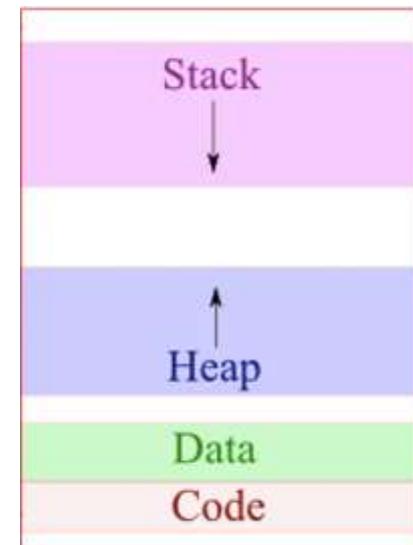
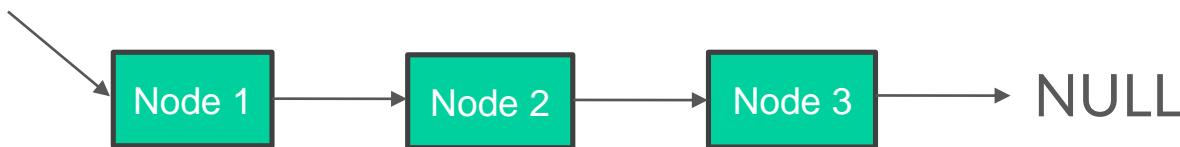
**Solution:** If you want your struct's data to be on the heap, you must deep copy any user-provided pointers (including strings) first

---

## Malloc example: ArrayList

Allocating new list nodes for a linked list as a user requests them.

Head



---

## Malloc example: ArrayList

An ArrayList data structure is an array that resizes as there is a need for more space. The data stored in our list will be dog structs.

```
typedef struct dog_t {
 char *name;
 int age;
} dog;
```

---

## Malloc example: ArrayList

We also need to make an arraylist struct to keep track of the list of dogs, and other helpful info:

```
typedef struct arrayList_t {
 dog *list;
 int size;
 int capacity;
} arrayList;
```

---

## Malloc example: ArrayList

Let's say the initial array has 16 elements. On the array initialization, we malloc space as follows:

```
dog *dog_array = (dog *)malloc(sizeof(dog) * 16);
```

---

## Malloc example: ArrayList

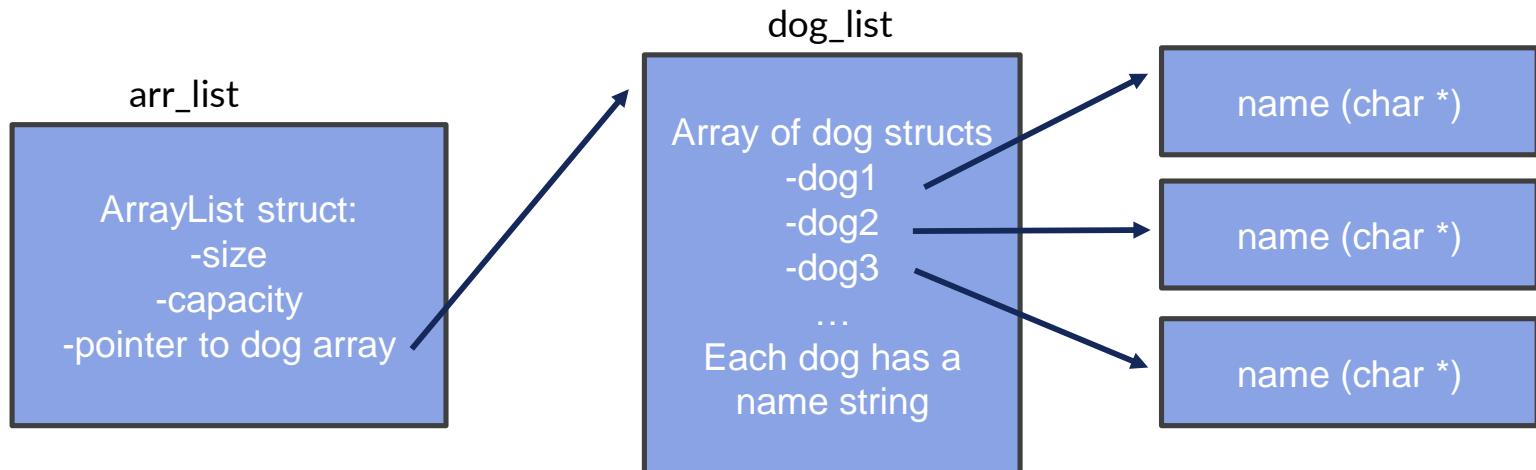
We can also assign the list we just allocated to an instance of the arraylist struct, along with an initial size and capacity:

```
arraylist *arr_list = malloc(sizeof(arraylist));
arr_list -> size = 0;
arr_list -> capacity = 16;
arr_list -> list = dog_array;
```

---

# Malloc example: ArrayList

ArrayList diagram (each block is allocated on the heap):



---

## Malloc example: ArrayList

How do we add an item to the back of the list?

- Index into the list at arr\_list -> size
- Initialize the dog struct's data
- Increase the size of the list

---

## Malloc example: ArrayList

How do we add an item to the back of the list?

```
//The dog is 1 year old
arr_list -> list[arr_list -> size].age = 1;
```

**// Does this work?**

```
arr_list -> list[arr_list -> size].name = "puppy"
```



## Malloc example: ArrayList

We don't want any data to be statically allocated, so when we assign the name to the dog that name itself should be malloc-ed

```
//You can malloc space for a string as follows
char *static_name = "puppy";
char *dyn_name = malloc(sizeof(char) * (strlen(static_name) + 1));
strcpy(dyn_name, static_name);
dyn_name[strlen(static_name)] = '\0'; //set null terminator
```

---

## Malloc example: ArrayList

Now back to adding the item to the list:

```
arr_list -> list[arr_list -> size].age = 1;
arr_list -> list[arr_list -> size].name = dyn_name;

arr_list -> size++;
```

---

## Malloc example: ArrayList

What do we do if the size of the array exceeds capacity? We can simply realloc to maintain the array values and increase the size.

```
int new_size = arr_list -> capacity * 2 * sizeof(dog);
arr_list -> list = realloc(arr_list->list, new_size);

//realloc takes in an old pointer and a new size
//copies data from old pointer into new block of memory of
requested size.
```

---

## Malloc example: ArrayList

How do we remove an element from the back of the list?  
Can we just hit subtract from the size and call it a day?

```
arr_list -> size--; //we're done! ...right?
```

---

## Malloc example: ArrayList

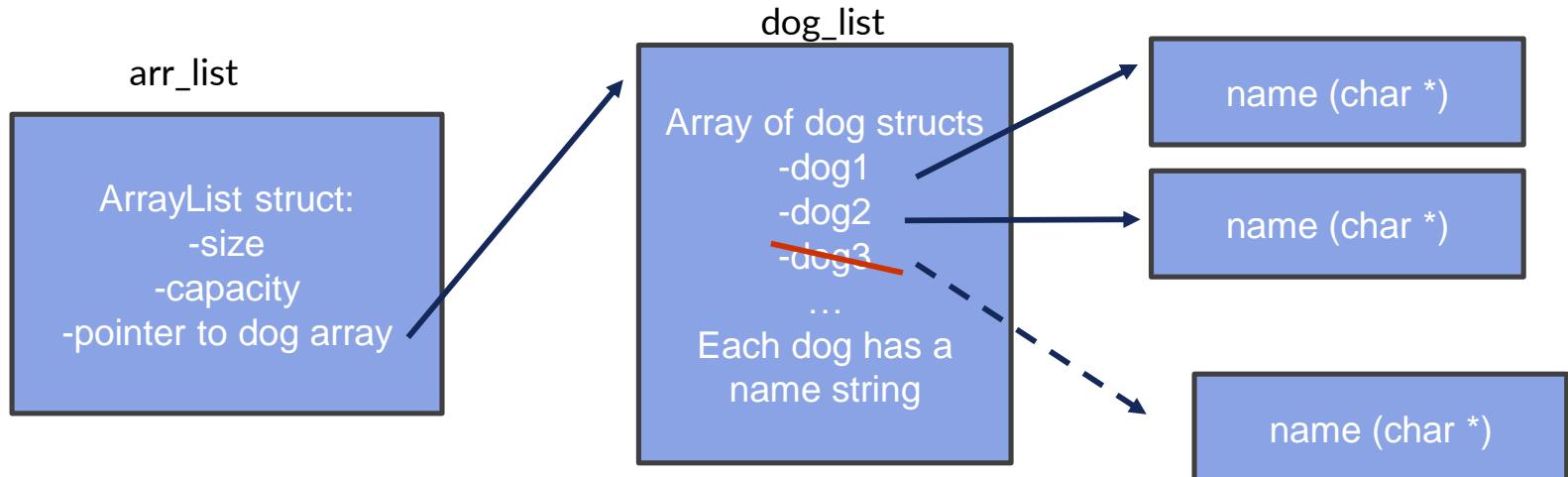
No! Remember, we allocated space for a string on the heap!

If we don't free that block of memory when it becomes unused, we get a  
*Memory Leak*

```
free(arr_list -> list[arr_list -> size - 1].name);
arr_list -> size--;
```

# Malloc example: ArrayList

Memory Leak diagram: when dog3 gets deleted, its name pointer is lost and so that piece of memory is never freed.



---

# Unions

---

# Unions

- Similar to structs, but all of the members are stored in the same spot in memory
- **Example:**

```
union Data {
 int i;
 float f;
 char str[20];
};
```

- ∅ What is the size of the union?

---

# Unions in C – Beware!

- Can only store 1 member's value at a time – watch out for corrupting values!
- What prints when this code is run?

```
union Data data;
data.i = 10;
data.f = 220.5;
strcpy(data.str, "C Programming");

printf("data.i : %d\n", data.i);
printf("data.f : %f\n", data.f);
printf("data.str : %s\n", data.str);
```

---

# Uses for Unions

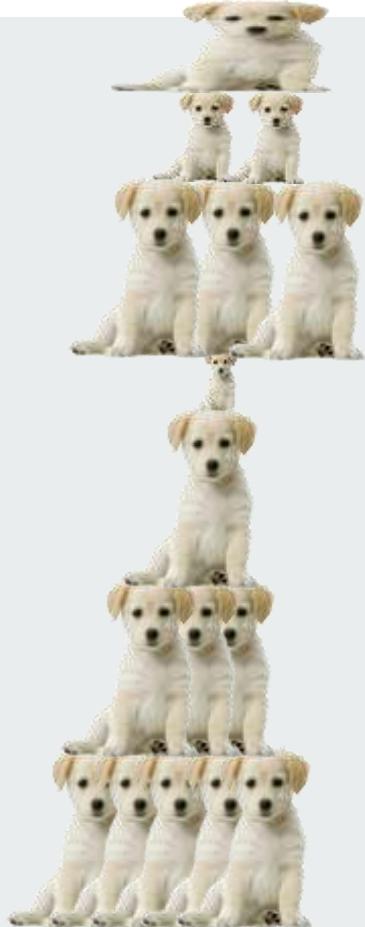
- Convert between the binary representations of different types; this can be useful for embedded programming when writing to registers
  - Note: this converts between the binary representations, not the values
  - For instance, using this to convert 5 to a float would not result in 5.0
- Can be used to implement pseudo-polymorphism
  - Use a union to hold the values that are different across the different subclasses
  - Have some other variable that determines which union field is valid
  - What if you need to hold multiple values for each “subclass”?
  - Used this way in HW9!
- Used to conserve memory when you only need one field

---

# CS 2110 - Lab 17

## Intro to Malloc Implementation

Monday, November 22<sup>nd</sup>, 2021



---

## Homework 9 – Linked List

- Apply dynamic memory knowledge in C
- Released!
- **Due Tuesday, November 30<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until Wednesday 12/1
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

# Homework 10 – Malloc Implementation

- Write your own implementation of malloc, calloc, realloc, and free!
- Will release Wednesday, December 1<sup>st</sup>
- **Due Tuesday, December 7<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until Wednesday 12/8
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

## Timed Lab 4

- Next Wednesday (December 1<sup>st</sup>) during lab
- You must come to class and take the TL here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TA(s)
- Full 75 minutes to complete the TL!
- Will cover HW7, HW8 and HW9 topics

---

## Lab Assignment: Canvas Quiz

- 1) Go to Quizzes on Canvas
- 2) Select Lab 17
- 3) Access code: **sbrk**
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)



---

## "Out" Variables

- How do you return multiple values from a function?
  - E.g. the "pop" functions in HW9
- Could use structs with one member for every return value
  - Adds lots of boilerplate
  - You'd need a new struct for every function returning multiple values
  - Can get confusing (you need to go to the header file/struct declaration to see what types are contained)

---

# "Out" Variables

- Solution: pass in a pointer to a local variable in the calling function!
- Pick one type as your "main" return value
- Return the rest of the values by writing to "out" variables

```
int main(void) {
 char *s; // hello will write to this
 int len = hello(&s);
 // now 's' is "hello"
 ...
}
```

Caller passes a pointer to a local variable

```
// Returns a string along with its length
int hello(char **out) {
 char *c = "hello";
 // how do we return 'c' in 'out'?
 return strlen(c);
}
```

Callee uses the pointer to modify the caller's local variable

---

# "Out" Variables

- Why doesn't this work? How can we fix it?

```
// Returns a string along with its length

int hello(char **out) {
 char *c = "hello";
 out = &c; // doesn't work!
 return strlen(c);
}
```

---

## "Out" Variables

- Changing the value of `out` rather than `*out` just modifies our own stack frame, not the caller's

```
// Returns a string along with its length
```

```
int hello(char **out) {
 char *c = "hello";
 *out = c;
 return strlen(c);
}
```

---

## Malloc & Friends

- `malloc(size)`: allocates `size` bytes on the heap
- `calloc(nmemb, size)`: allocates `nmemb*size` bytes on the heap and zeroes them out
- `realloc(ptr, size)`: the “resize” function; returns a block of `size` bytes with the data from `ptr`
- These functions return a `void*` to the new block, or `NULL` on failure

---

## Free

- `free(ptr)`: Gives a pointer returned by `malloc` (or friends) back to the heap; we no longer need it
- **Never use a freed pointer, it is no longer valid.**
- Failing to free pointers you're not using will cause memory leaks; the heap will run out of space
- `realloc(ptr, size)` will also free `ptr`, so you should use the pointer returned instead

---

## What do we need to do?

Our implementation of the `malloc` library must:

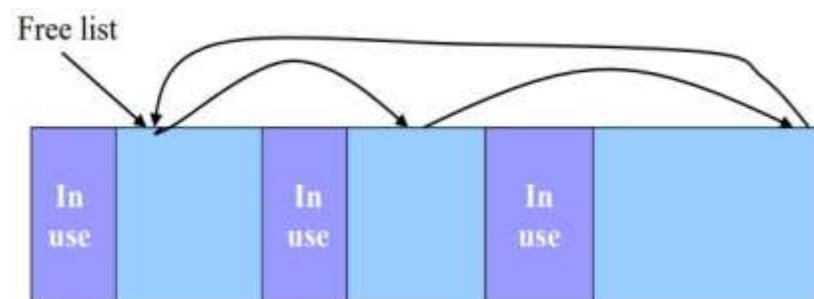
- Give out free memory as requested
- Take memory back when it's freed
- Request more free memory from the operating system when we don't have enough

How can we keep track of our free memory?

---

## The Freelist!

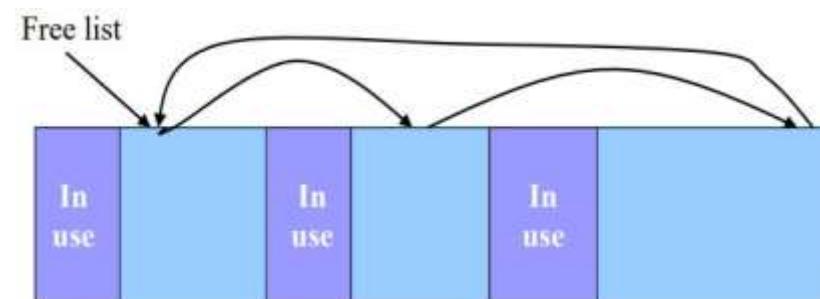
- We need a way to keep track of available memory, which could be all over the heap
- Solution: a linked list of all available memory blocks
  - Specifically, blocks that are NOT currently allocated



---

## The Freelist!

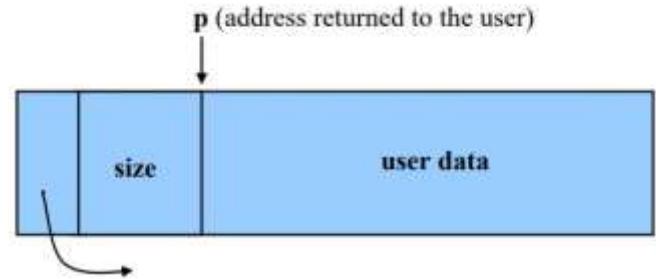
- When `malloc()` is called, we search through freelist to find block of adequate size to return to the user
- How do we know the size of each block in memory?



---

## Metadata

- Each block has “metadata”—useful information about the block
- We need to keep track of size of the block (why?)
- We also need a pointer to the next node in the list
- The address returned to user is after the metadata



---

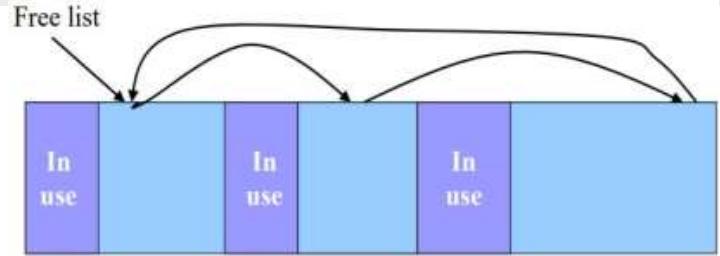
## Malloc Implementation

With the freelist, we are ready to see how malloc works:

1. User asks for block of certain size
2. malloc looks through freelist for this size
3. If we find such a block, remove it from the freelist and return it to the user!

---

## Malloc Implementation

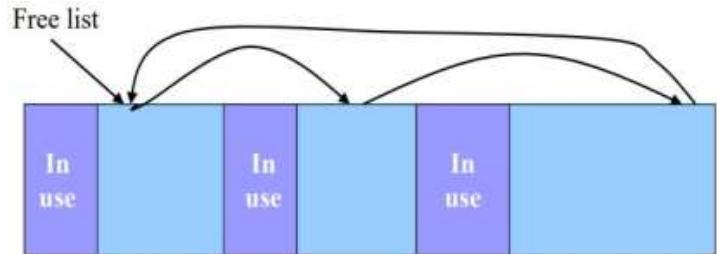


What if we don't find a block of the right size?

- If we have a block greater than the size:
  - Divide the block into two parts, one of the correct size, and one with whatever's left over
  - Remove and return new block with proper size
- What if we don't have any blocks that are big enough?

---

# Malloc Implementation



- `sbrk()`
  - System call that gives us a large chunk of heap space to work with
  - We can add this new block to freelist and continue
- `sbrk` returns -1 in case of error
  - If this occurs, we have run out of heap space
  - We should simply return NULL

---

## Other functions

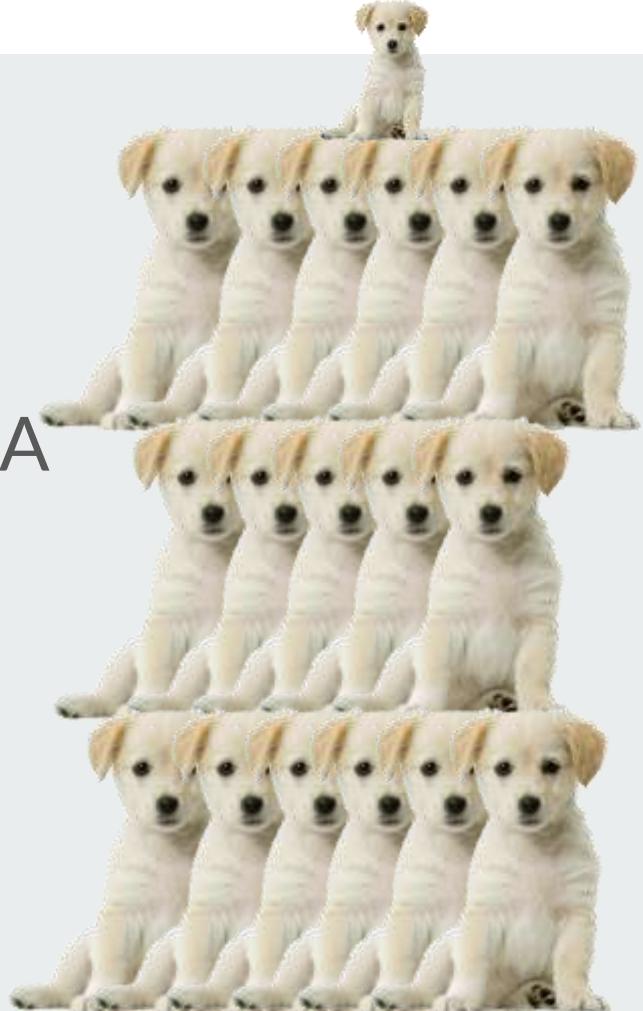
- How can we free a block?
- Can we implement `calloc()` and `realloc()`?
  - Don't reinvent the wheel – use `malloc()` and `free()` in your implementations of these functions!

---

# CS 2110 - Lab 18

Malloc Implementation and Q&A

Monday, November 29<sup>th</sup>, 2021



---

## Homework 9 – Linked List

- Apply dynamic memory knowledge in C
- Released!
- **Due Tuesday, November 30<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until Wednesday 12/1
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

# Homework 10 – Malloc Implementation

- Write your own implementation of malloc, calloc, realloc, and free!
- Will release Wednesday, December 1<sup>st</sup>
- **Due Tuesday, December 7<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- Standard grace period until Wednesday 12/8
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

## Timed Lab 4

- This Wednesday (December 1<sup>st</sup>) during lab
- You must come to class and take the TL here
- Open book, open note, open internet
  - Use one device only: transfer any notes to your laptop
  - No communication with anyone else except your TA(s)
- Full 75 minutes to complete the TL!
- Will cover HW7 (string manipulation), HW8 (GBA concepts, particularly videoBuffer) and HW9 topics (malloc and friends)

---

## Final Review Lab

- Next Monday (December 6<sup>th</sup>) during lab
- Review of the whole course before the final – you **do not** want to miss this lab
- Please comment on pinned Ed post #1004 to decide which topics will be focused on!

---

## CIOS is open!

- Please fill it out! Your feedback is extremely useful for improving the course
- <http://gatech.smartevals.com> (or linked in Canvas)
- Don't ask about a CIOS Bonus yet! If we decide to do a CIOS incentive, it will be announced via Canvas at a later date.

---

# A Note from Shawn Wahi

Hello!

If you have time and motivation, please consider filling out CIOS for me too. Historically the Head TA gets significantly less CIOS comments compared to lab TAs. Since CIOS is fully anonymous, feel free to provide an honest comments (positive or negative). Let me know what things I did well and what areas you feel I need to improve upon. I'll be sure to read every comment I receive during winter break, while sipping on some hot chocolate :)

Your favorite Head TA,  
Shawn Wahi

P.S. - I'd rather get negative CIOS comments than no CIOS comments (at least it'd be entertaining to read).

---

## Thank a Teacher

- Let your TA, Head TA, or professor know that they did an outstanding job!
- Help them get a sense of the positive impact they've had on your learning this semester ☺
- <http://thankateacher.gatech.edu>

---

## Lab Assignment: Canvas Quiz



- 1) Go to Quizzes on Canvas
- 2) Select Lab 18
- 3) Access code: **press\_f\_for\_canaries**
- 4) Get 100% to get checked off!
  - a) Unlimited attempts
  - b) Collaboration is **allowed!**
  - c) Ask your TAs for help :)

---

# Freelist — Review

---

## What do we need to do?

Our implementation of the malloc library must:

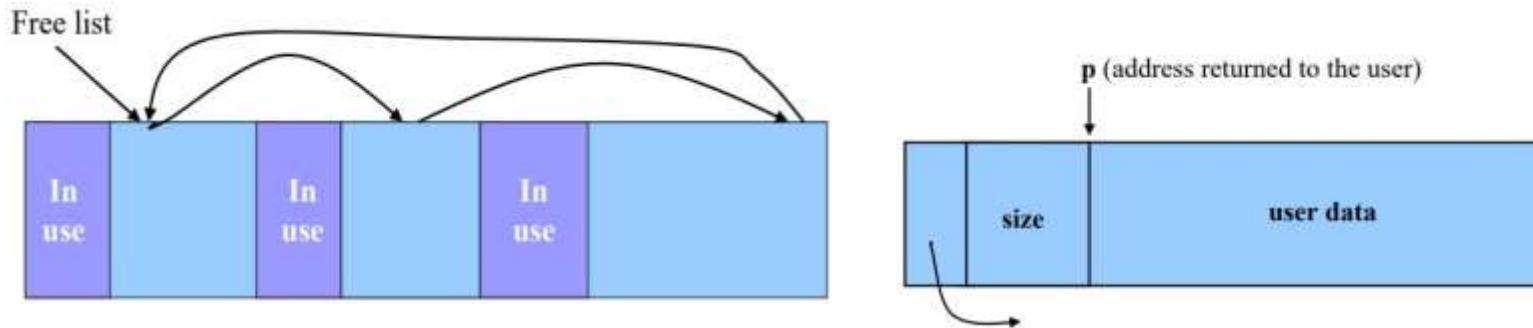
- Give out free memory as requested
- Take memory back when it's freed
- Request more free memory from the operating system when we don't have enough

We keep track of our memory in the *freelist*.

---

## The Freelist!

- The freelist is a linked list of blocks that are *free* (i.e. not in use)
- Each block has metadata (information about the size of the block and a pointer to the next node)



---

## Basic Malloc Implementation

When the user calls malloc:

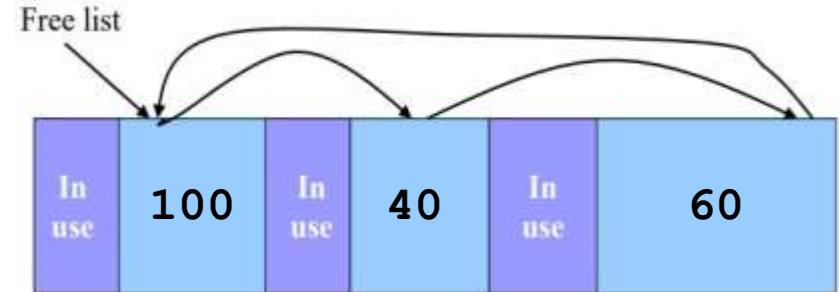
1. User asks for block of certain size
2. malloc looks through freelist for this size
3. If we find such a block, remove it from the freelist  
and return it to the user!
4. Otherwise, we split a larger block up, or request  
more space using sbrk

---

# More Malloc Implementation!

---

## Matching



- If the user requests 60 bytes of space, which block should be returned?
- The “best match” choice is to return the last block, but this requires iterating through the whole list to see that it exists
- Another option would be to do a “first match”—we could split the block of 100 into blocks of 40 and 60

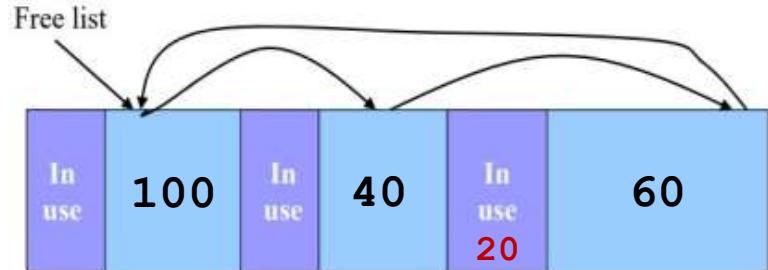
---

## Homework 10 Implementation

- Our freelist is a singly-linked, non-circular linked list
- Blocks are ordered by memory address
  - i.e., the first node in the linked list will be the block starting at the lowest address
- “Best match”—we will always iterate through the freelist until we find a block of the correct size

---

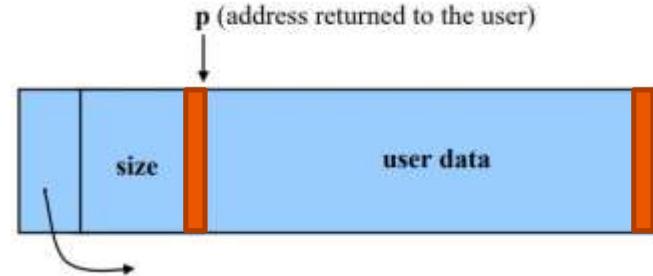
## Free Semantics



- What if the user were to free the block of size 20?
- We might end up with three adjacent blocks (40, 20, 60)—what if the user then requests 100 bytes?
  - Solution: we should merge these nodes after freeing
- How can we tell if two nodes are adjacent?
  - Hint: our freelist is ordered by memory address

---

## Canaries (not on HW10!)



- How can we ensure that the user isn't writing outside of the block given to them?
- Solution: canaries!
- We write some pre-defined (potentially random) value to each side of the user data block, and then check that they are correct when the user calls free

# Q&A Time!

Ask us any questions you have (malloc-related or otherwise)

Having a good theoretical foundation is the key to completing HW10 quickly





# CS 2110 - Lab 19

Final Exam Review

Monday, December 6<sup>th</sup>, 2021



---

# Homework 10 – Malloc Implementation

- Write your own implementation of malloc, calloc, realloc, and free!
- Released Wednesday, December 1<sup>st</sup>
- **Due Tuesday, December 7<sup>th</sup> at 11:59 PM**
- Files available on Canvas
- Submit on Gradescope (unlimited submissions)
- No late penalty if you submit by Wednesday, December 8th at 11:59 PM
- Extensions for non-emergency/non-medical reasons are not possible on HW9 or HW10.

---

## CIOS is open!

- Please fill it out! Your feedback is extremely useful for improving the course
- <http://gatech.smartevals.com> (or linked in Canvas)
- Don't ask about a CIOS Bonus yet! If we decide to do a CIOS incentive, it will be announced via Canvas at a later date.

---

## Another Note from Shawn Wahi

Hello!

I'm back following my debut note in last week's slide deck. For the final lab, I wanted to ask everyone to take a moment to read Ed Discussion post #1134 titled "A Public Thank You Note to My TA Team (From Shawn Wahi)." My TA team had no advance notice of me writing this post. I just felt that everyone on my team has put in so much effort into making this the best possible version of the course and they rightfully deserve recognition. Don't hesitate to reply to the post and thank the TAs as well. I couldn't have orchestrated this semester without their continued help and support. ☺

- Shawn Wahi

---

## Thank a Teacher

- Let your TA or professor know that they did an outstanding job!
- Help them get a sense of the positive impact they've had on your learning this semester ☺
- <http://thankateacher.gatech.edu>

---

## **Need to make up an assessment?**

- Email Shawn Wahi ASAP
- Wednesday, December 8, 2021 is the last chance to make up a Quiz or Timed Lab
- It is too late to make up any missed demos for the semester

---

## Finalized Semester Grades

- All semester grades (excluding the Final Exam) will be finalized by Thursday at 9am
- Check your grades Thursday morning and check if anything is incorrect. If so, email Shawn Wahi immediately

---

# Final Exam

- Section C: Thursday, December 9<sup>th</sup>, 2:40pm–5:30pm
  - Clough 152
- Section B: Tuesday, December 14<sup>th</sup>, 2:40–5:30pm
  - Clough 144
- **Takes place in your usual lecture hall**
- TAs will be present to answer questions, like with other assessments

---

# Final Exam

- When the exam block starts, the following will open on Canvas:
  - A quiz called “Final Exam”
  - An assignment called “Final Exam: LC-3 Assembly Coding”
  - An assignment called “Final Exam: C Coding”
- There will be autograded Gradescope assignments for the coding questions, as well as local autograders
- No content from lectures prefixed "Misc C topics: " will be tested on the Final Exam
- Need to know all lab content through the end of the semester

---

## Final Exam

- The quiz portion is expected to take about 60% of the time, and the timed lab portion about 40% combined
  - You may divide your time however you like
  - The entire block (2 hours, 50 minutes) will be available
- The final must be taken in-person
- It will be open-note, open-resource, etc.
  - No communicating with other people
  - One device only

---

So....

- Did you figure out the pattern with the dogs on the start of every lab slide deck?
- If not, it's time for the lab TA to spoil the semester long easter egg :)



---

## Short Survey

- Please take the next five minutes to complete the following short survey regarding some of the new strategies and logistics we piloted this semester
- All replies are anonymous
- Survey Link: <https://b.gatech.edu/2ZYgvpg>

---

# Review topics!

- 2's complement addition
- Bit masking
- Logic gates (transistor-level)
- Decoder/MUX, adder
- Three sequential logic latches
- Truth tables
- Gray code and K-maps
- LC-3 datapath/control signals
- Special-instruction-based I/O
- Polling vs. interrupt I/O
- The stack, subroutines and calling convention
- Pointer arithmetic
- GBA videoBuffer and colors
- How DMA works
- Heap (Malloc & friends)
- Freeing things
  - Structs containing structs, strings, pointers and unions
- Whatever you want!